DEPARTAMENTO DE INFORMÁTICA

Trabalho de Sistemas de Base de Dados
SQL Server 2008

2009/2010

Autores:
Diogo André Ribeiro Mourão nº31213
João André Figueiredo Gonçalves Saramago nº32461
Pedro Rafael Pereira Martins nº31227

Conteúdo

1 Introdução **5**
- 1.1 Estrutura . . . 5
- 1.2 História e Aplicabilidade . . . 6
- 1.3 Notas sobre instalação do sistema . . . 8

2 Cobertura do SQL **9**
- 2.1 DDL . . . 9
  - 2.1.1 Tabelas . . . 9
  - 2.1.2 Triggers . . . 10
- 2.2 DML . . . 12
  - 2.2.1 Principais Clausulas . . . 12
- 2.3 Tipos . . . 20
  - 2.3.1 Numéricos Exactos . . . 20
  - 2.3.2 Data e Hora . . . 20
  - 2.3.3 Strings Unicode . . . 21
  - 2.3.4 Strings Binárias . . . 21
  - 2.3.5 Strings . . . 21
  - 2.3.6 Outros tipos de dados . . . 21

3 Armazenamento e file structure **22**
- 3.1 Gestão de memória . . . 22
- 3.2 Ficheiros e páginas . . . 24
- 3.3 Extensões e filegroups . . . 26
- 3.4 Estrutura de dados e partições . . . 28
- 3.5 Recuperação de dados . . . 31

4 Indexação **32**
- 4.1 Sintaxe . . . 32
  - 4.1.1 Criação de Índices Relacionais[11] . . . 32
  - 4.1.2 Criação de Índices XML[13][21] . . . 35
  - 4.1.3 Criação de Índices *SPATIAL*[12][32] . . . 38
  - 4.1.4 Criação de Índices *FULLTEXT*[10][5][18][33] . . . 41
  - 4.1.5 Remoção de Índices[17] . . . 43

4.1.6 Remoção de Índices *FULLTEXT*[16] 43
4.2 Estruturas de Dados 44
4.2.1 *CLUSTERED Index*[8] 44
4.2.2 *NONCLUSTERED Index*[24] 45

5 Processamento e optimização de perguntas[6] 46
5.0.3 Query Optimizer 47
5.1 Algoritmos implementados 49
5.1.1 Junção 49
5.2 Mecanismos para expressões complexas 53
5.2.1 Paralelismo[4] 53
5.2.2 Materialização 53
5.2.3 Pipelining 54
5.3 Estatísticas[6] 55
5.3.1 Tipo de Estimativas 55
5.3.2 Algumas considerações 56
5.3.3 Exemplos 56
5.4 Caching de planos de execução[5] 59
5.4.1 Exemplo Adhoc Workload 61
5.4.2 Exemplo Prepared Workload 62
5.5 Mecanismos para ver planos de execução 64
5.5.1 Graficamente 64
5.5.2 Formato XML 65
5.5.3 Texto 66

6 Gestão de transacções e controlo de concorrência 67
6.1 Modos de transacção 68
6.1.1 Transacções AutoCommitted[34] 68
6.1.2 Transacções Implícitas[36] 68
6.1.3 Transacções Explicitas[35] 70
6.2 Tópicos avançados sobre Transacções 71
6.2.1 Transacções Nested[37] 71
6.2.2 Transacção de longa duração[38] 71
6.2.3 Savepoints[29] 71
6.2.4 Instruções T-SQL permitidas em Transacções[22] . . . 72
6.3 Protocolos de isolamento 73
6.3.1 READ UNCOMMITTED[27] 74
6.3.2 READ COMMITTED[26] 74
6.3.3 REPEATED READ[28] 74
6.3.4 SNAPSHOT[31] 74
6.3.5 SERIALIZABLE[30] 75
6.3.6 Row versioning 75
6.4 Modos de Lock[23] 76
6.4.1 Shared locks 76

6.4.2 Exclusive Locks . . . 76
6.4.3 Update Locks . . . 76
6.4.4 Intent Locks . . . 77
6.4.5 Schema Locks . . . 77
6.4.6 Key-range Locks . . . 77
6.5 Deadlocks[14] . . . 78
6.5.1 Detecção de deadlocks e terminação de deadlocks[15] . . . 78
6.5.2 Detecção de deadlock e escolha da vitima . . . 79
6.6 Granularidade[19] . . . 80

7 Suporte para bases de dados distribuídas 81
7.1 Base de Dados homogénea e heterogénea . . . 81
7.2 Replicação . . . 82
7.3 Fragmentação . . . 88
7.4 Bases de dados Heterogéneas . . . 88
7.5 Transparência de dados . . . 90
7.6 Transacções distribuídas . . . 90
7.7 Resolução de conflitos . . . 91
7.8 Locks globais . . . 92
7.9 Propagação de actualizações . . . 92
7.10 Formas restritivas de uso . . . 93

8 Outras características do sistema estudado 94
8.1 Integração com o CLR[7][25][9] . . . 94
8.1.1 *Stored Procedures* em CLR . . . 95
8.1.2 *Triggers* em CLR . . . 96
8.1.3 *User Defined Types* em CLR . . . 96
8.2 XML[20] . . . 98

Lista de Figuras

3.1 *Organização de páginas em ficheiros* 25
3.2 *Organização de páginas* 26
3.3 *Tipos de extensões* 27
3.4 *Gestão de extensões* 27
3.5 *Exemplo da organização de uma BD* 28
3.6 *Arquitectura de uma tabela* 28
3.7 *Páginas IAM em heaps* 30
3.8 *Índices não clustered* 30

4.1 *Clustered Index* 44
4.2 *NONCLUSTERED Index* 45

5.1 *Passos para criação dos planos de execução* 46
5.2 *Passos do Optimizer* 48
5.3 *Hash Join* 50
5.4 *Comando updatestats* 57
5.5 *Comando autostats* 57
5.6 *Verificar se a criação de estatísticas esta ligada.* 58
5.7 *Adhoc* 61
5.8 *Adhoc* 62
5.9 *Prepared* 63
5.10 *Visualização gráfica* 64
5.11 *XML* 65
5.12 *Texto* 66

6.1 *Deadlock* 78

7.1 *Sistema de bases de dados distribuídas em localizações distintas* 82
7.2 *Topologia de replicação* 83
7.3 *Replicação transaccional* 86
7.4 *Replicação Merge* 87
7.5 *Replicação snapshot* 87
7.6 *Arquitectura de ligação dos servidores* 89

Capítulo 1

Introdução

Este trabalho foi desenvolvido no âmbito da disciplina de Sistemas de Bases de Dados onde a organização e implementação deste tipo de sistemas foram estudadas ao detalhe. Durante as aulas a matéria dada foi acompanhada pelo estudo do sistema Oracle dentro dos temas falados. O objectivo deste trabalho era analisar outro sistema de gestão de base de dados sem ser o Oracle 10g, dentro da matéria estudada no âmbito da disciplina. Um sistema de gestão de base de dados tem como intuito deixar a gestão, manipulação e organização de informação dentro do sistema a cargo deste, sendo fácil para o utilizador o acesso e todo o tipo de alterações a dados, sem que seja necessários grandes conhecimentos do funcionamento interno do sistema. O sistema de base de dados que foi eleito por nós para analisarmos foi o Microsoft SQL Server 2008. Esta escolha foi feita por este sistema funcionar em Windows de maneira simples e pois tínhamos bastante informação sobre as várias temáticas a desenvolver no trabalho.

1.1 Estrutura

Este trabalho está organizado para que a informação demonstrada seja sequencial, ou seja, toda a informação referida já foi explicada anteriormente ou está a ser desenvolvida nesse capítulo. Inicialmente daremos uma pequena introdução histórica deste sistema de bases de dados e onde este é aplicável, bem como um pequeno manual de instalação deste sistema. De seguida será descrita a cobertura que o SQL Server faz do standard de SQL, especificando a sua cobertura ao nível de DML e DDL. Após esta explicação inicial iremos entrar dentro da especificação e implementação do sistema em concreto. Começamos por explicar todo o mecanismo de armazenamento de dados e como estes são organizados em ficheiros, de maneira a serem acedidos, alterados, adicionados e removidos. Seguidamente foram enumerados todos os mecanismos que o SQL Server proporciona para indexação e hashing, bem como as estruturas de dados que os suportam. O processamento

e optimização de perguntas é o próximo tema abordado no trabalho, sendo descritos os mecanismos que servem para este propósito dentro do sistema. A gestão de transacções e controlo de concorrência em sistemas SQL Server é abordada a seguir, com a explicação de como é feita e que mecanismos suportam estas características. De seguida descrevemos a implementação de bases de dados distribuídas no SQL Server, apontando como pode ser feita, as suas características essenciais e toda a implementação que suporta este sistema. Para finalizar falamos de algumas características que achamos interessantes dentro do SQL Server, bem como de ferramentas que poderão estar associadas ao sistema.

1.2 História e Aplicabilidade

O código base do SQL Server, até à versão 7.0, tem como origem o Sybase SQL Server, e foi o começo da Microsoft no mercado das base de dados empresariais, competindo com a Oracle, IBM e posteriormente a Sybase. A Microsoft, a Sybase e a Ashton-Tate uniram-se em 1989 para criar a primeira versão, chamada SQL Server 1.0, para o OS/2, que foi uma implementação do Sybase SQL Server 3.0 para UNIX. O Microsoft SQL Server 6.0 foi a primeira versão pata Windows NT sem a participação da Sybase.

No tempo do Windows NT, a Sybase e a Microsoft separaram-se e cada um desenvolveu o seu sistema de base de dados. A Microsoft renegociou os direitos exclusivos de todas as versão do SQL Server para o Windows. Mais tarde, a Sybase mudou o nome do seu sistema de base de dados para Adaptive Server Enterprise para não haver confusões com o sistema da Microsoft. Até 1994, o Sql Server da Microsoft trazia três avisos de copyright da Sybase como indicação de origem.

Desde da separação, várias revisões ao sistema foram feitas de forma independente. O SQL Server 7.0 foi reescrito baseado no código da Sybase. Este foi sucedido por o SQL Server 2000, que foi a primeira versão com uma variante para a arquitectura IA-64.A partir dessa existiram mais duas versões do SQL Server, a de 2005 e a de 2008 que é a actual.

Versão	Ano	Release Name	Codename
1.0 (OS/2)	1989	SQL Server 1.0	-
4.21 (WinNT)	1993	SQL Server 4.21	-
6.0	1995	SQL Server 6.0	SQL95
6.5	1996	SQL Server 6.5	Hydra
7.0	1998	SQL Server 7.0	Sphinx
-	1999	SQL Server 7.0 OLAP Tools	Plato
8.0	2000	SQL Server 2000	Shiloh
8.0	2003	SQL Server 2000 64-bit Edition	Liberty
9.0	2005	SQL Server 2005	Yukon
10.0	2008	SQL Server 200	Katmai

O SQL Server é um sistema de gestão de bases de dados com uma enorme escalabilidade estando disponível para todo o tipo de máquinas desde portáteis até servidores empresariais que TB de dados. Existe também a compatibilidade com o sistema PocketPC, permitindo a sua utilização em PDA's e mecanismos com essa aplicação incorporada. Este sistema tem a sua principal aplicabilidade em suportar bases de dados, que podem ser distribuídas, de forma a guardar toa a informação necessária.

1.3 Notas sobre instalação do sistema

Instalar este Sistema de Base de Dados é muito simples e não exige nenhum skill em especial. Portanto remetemos a informação de instalação para um tutorial, com imagens incluidas, num site WEB.

O site com o tutorial de instalação é o seguinte:
`http://www.packtpub.com/article/microsoft-sql-server-2008-installation-made-easy`

Capítulo 2

Cobertura do SQL

2.1 DDL

A DDL ou Data Definition Language é uma linguagem usada para definição de estrutura de dados. No SQL as instruções DDL têm como objectivo definir o modelo de dados e o modelo entidade-relações de uma Base de Dados. As instruções DDL que manipulam tabelas são as seguintes: **CREATE**, **DROP** e **ALTER**.

2.1.1 Tabelas

Cláusula **CREATE**

A sintaxe simplificada para a instrução CREATE é:

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ]
        table_name
        ( { <column_definition> | <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] )
    [ ON { partition_scheme_name ( partition_column_name ) |
        filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } ]
[ ; ]
```

Para mais detalhes na criação de tabelas, consulte `http://msdn.microsoft.com/en-us/library/ms174979.aspx`

Cláusula **DROP**

```
DROP TABLE
    [ database_name . [ schema_name ] . | schema_name . ]
        table_name ]
```

Cláusula **ALTER**

A sintaxe simplificada da instrução ALTER é:

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name .
    ] table_name
{
    ALTER COLUMN column_name
    {
        [ type_schema_name. ] type_name [ ( { precision [ ,
            scale ]
            | max | xml_schema_collection } ) ]
        [ COLLATE collation_name ]
        [ NULL | NOT NULL ]
    | {ADD | DROP } { ROWGUIDCOL | PERSISTED | NOT FOR
        REPLICATION}
    }
    | [ WITH { CHECK | NOCHECK } ] ADD
    {
        <column_definition>
      | <computed_column_definition>
      | <table_constraint>
    } [ ,...n ]
    | DROP
    {
        [ CONSTRAINT ] constraint_name
        [ WITH ( <drop_clustered_constraint_option> [ ,...n ] )
            ]
        | COLUMN column_name
    } [ ,...n ]
    | [ WITH { CHECK | NOCHECK } ] { CHECK | NOCHECK }
        CONSTRAINT
        { ALL | constraint_name [ ,...n ] }
    | { ENABLE | DISABLE } TRIGGER
        { ALL | trigger_name [ ,...n ] }
    | SWITCH [ PARTITION source_partition_number_expression ]
        TO target_table
        [ PARTITION target_partition_number_expression ]
}
[ ; ]
```

Para mais detalhes na alteração de tabelas, consulte `http://msdn.microsoft.com/en-us/library/ms190273.aspx`

2.1.2 Triggers

O SQL Server suporta Triggers. Um trigger é um procedimento executado em resposta a algum evento. Existem instruções para criação, alteração e remoção de triggers.

Para criar triggers, existe a instrução:

```
CREATE TRIGGER trigger_name
ON table_name
{ FOR | AFTER | INSTEAD OF }
```

```
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
{PROCEDURE | RAISERROR}
GO
```

Para alterar triggers, existe a instrução:

```
ALTER TRIGGER trigger_name
ON table_name
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
AS
{PROCEDURE | RAISERROR}
GO
```

Para remover triggers, existe a seguinte instrucção:

```
DROP TRIGGER trigger_name
```

Para informações mais detalhadas sobre Triggers, consulte `http://msdn.microsoft.com/en-us/library/ms189833.aspx`.

2.2 DML

A DML ou Data Manipulation Language é um conjunto de linguagens que manipulam dados em Bases de Dados. Permitem que utilizadores insiram dados, removam dados, actualizem dados e obtenham dados de uma Base de Dados. É importante referir que o DML manipula apenas os dados, e não o modelo de dados e o modelo relação-entidades. No SQL as principais instruções DML são o **SELECT**, **INSERT**, **UPDATE** e **DELETE**.

2.2.1 Principais Clausulas

Cláusula **SELECT**:

```
SELECT statement ::=
    [WITH <common_table_expression> [,...n]]
    <query_expression>
    [ ORDER BY { order_by_expression | column_position [ ASC |
       DESC ] }
  [ ,...n ] ]
    [ COMPUTE
  { { AVG | COUNT | MAX | MIN | SUM } ( expression ) } [ ,...n ]
  [ BY expression [ ,...n ] ]
    ]
    [ <FOR Clause>]
    [ OPTION ( <query_hint> [ ,...n ] ) ]


<query_expression> ::=
    { <query_specification> | ( <query_expression> ) }
    [  { UNION [ ALL ] | EXCEPT | INTERSECT }
        <query_specification> | ( <query_expression> ) [...n ] ]

<query_specification> ::=
SELECT [ ALL | DISTINCT ]
    [TOP expression [PERCENT] [ WITH TIES ] ]
    < select_list >
    [ INTO new_table ]
    [ FROM { <table_source> } [ ,...n ] ]
    [ WHERE <search_condition> ]
    [ GROUP BY [ ALL ] group_by_expression [ ,...n ]
                [ WITH { CUBE | ROLLUP } ]
    ]
    [ HAVING < search_condition > ]
```

Os argumentos mais importantes da cláusula **SELECT** estão detalhados abaixo:

- ALL - Especifica que linhas duplicadas aparecem no resultado da pergunta.
- DISTINCT - Especifica que linhas duplicas não aparecem no resultado.

- TOP expression - Indica que apenas um certo número de resultados ou uma percentagem de resultados da consulta serão retornados.
- <select list> - Especifica quais as colunas seleccionadas para o resultado da consulta.
- * - Todos os atributos de todas as tabelas referidas serão mostrados.
- table_name | view_name | table_alias.* - Indica qual a tabela pela qual queremos mostrar todos os seus atributos no resultado da consulta.
- column_name - Indica a coluna que aparecerá no resultado da consulta.
- GROUP BY - Especifica como os grupos de dados serão colocados no resultado retornado.
- HAVING - Especifica uma condição de pesquisa para um grupo de dados. É tipicamente usado em conjunto com a cláusula GROUP BY.
- OPTION - Especifica que query hint deverá ser utilizada durante a execução da query. A sintaxe desta opção em particular é a seguinte:

```
<query_hint > ::=
{ { HASH | ORDER } GROUP
   | { CONCAT | HASH | MERGE } UNION
   | { LOOP | MERGE | HASH } JOIN
   | FAST number_rows
   | FORCE ORDER
   | MAXDOP number_of_processors
   | OPTIMIZE FOR ( @variable_name { UNKNOWN | =
     literal_constant } [ , ...n ] )
   | OPTIMIZE FOR UNKNOWN
   | PARAMETERIZATION { SIMPLE | FORCED }
   | RECOMPILE
   | ROBUST PLAN
   | KEEP PLAN
   | KEEPFIXED PLAN
   | EXPAND VIEWS
   | MAXRECURSION number
   | USE PLAN N'xml_plan'
     | TABLE HINT ( exposed_object_name [ , <table_hint> [
       [, ]...n ] ] )
<table_hint> ::=
[ NOEXPAND ] {
    INDEX ( index_value [ ,...n ] ) | INDEX = ( index_value
      )
   | FASTFIRSTROW
   | FORCESEEK
   | HOLDLOCK
   | NOLOCK
```

```
    | NOWAIT
    | PAGLOCK
    | READCOMMITTED
    | READCOMMITTEDLOCK
    | READPAST
    | READUNCOMMITTED
    | REPEATABLEREAD
    | ROWLOCK
    | SERIALIZABLE
    | TABLOCK
    | TABLOCKX
    | UPDLOCK
    | XLOCK
}
```

Cláusula **INSERT**:

```
[ WITH <common_table_expression> [ ,...n ] ]
INSERT
    [ TOP ( expression ) [ PERCENT ] ]
    [ INTO]
    { <object> | rowset_function_limited
      [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
    }
{
    [ ( column_list ) ]
    [ <OUTPUT Clause> ]
    { VALUES ( { DEFAULT | NULL | expression } [ ,...n ] )
    | derived_table
    | execute_statement
    }
}
    | DEFAULT VALUES
[; ]

<object> ::=
{
    [ server_name . database_name . schema_name .
      | database_name .[ schema_name ] .
      | schema_name .
    ]
        table_or_view_name
}
```

Os argumentos mais importantes da cláusula **INSERT** estão detalhados abaixo:

- TOP expression - Especifica o número ou percentagem de tuplos aleatórios que serão inseridos.
- INTO - É uma palavra opcional que é usada entre a instrução SELECT e o nome da tabela para onde se quer inserir tuplos.

- server_name - Nome do servidor onde a tabela está guardada.
- database_name - Nome da base de dados que contém a tabela.
- table_or view_name - Nome da tabela ou nome da vista para a qual queremos inserir os tuplos.
- VALUES - Especifica os valores dos atributos do tuplo a inserir.
- WITH (<table_hint_limited> [... n]) - Especifica as opções que podemos permitir para a tabela de destino. Um exemplo disso, é especificar qual o recurso bloqueado aquando da inserção. (p.e: TABLOCK). A sintaxe para esta opção é a seguinte:

```
<table_hint_limited> ::=
{
      KEEPIDENTITY
    | KEEPDEFAULTS
    | FASTFIRSTROW
    | HOLDLOCK
    | IGNORE_CONSTRAINTS
    | IGNORE_TRIGGERS
    | NOWAIT
      | PAGLOCK
    | READCOMMITTED
    | READCOMMITTEDLOCK
    | READPAST
    | REPEATABLEREAD
    | ROWLOCK
    | SERIALIZABLE
    | TABLOCK
    | TABLOCKX
    | UPDLOCK
    | XLOCK
}
```

Cláusula **UPDATE**:

```
[ WITH <common_table_expression> [...n] ]
UPDATE
     [ TOP ( expression ) [ PERCENT ] ]
     { <object> | rowset_function_limited
      [ WITH ( <Table_Hint_Limited> [ ...n ] ) ]
     }
     SET
          { column_name = { expression | DEFAULT | NULL }
            | { udt_column_name.{ { property_name = expression
                                    | field_name = expression }
                                  | method_name ( argument [ ,...n
                                      ] )
                                }
```

```
            }
          | column_name { .WRITE ( expression , @Offset ,
              @Length ) }
          | @variable = expression
          | @variable = column = expression [ ,...n ]
        } [ ,...n ]
    [ <OUTPUT Clause> ]
    [ FROM { <table_source> } [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                      | cursor_variable_name
                  }
                ]
              }
            }
    ]
    [ OPTION ( <query_hint> [ ,...n ] ) ]
[ ; ]

<object> ::=
{
    [ server_name . database_name . schema_name .
    | database_name .[ schema_name ] .
    | schema_name .
    ]
        table_or_view_name
}
```

Os argumentos mais importantes da cláusula **UPDATE** estão explicados na cláusula **INSERT** anteriormente apresentada. Não existe nenhum argumento de acréscimo mais importante.

Cláusula **DELETE**

```
[ WITH <common_table_expression> [ ,...n ] ]
DELETE
    [ TOP ( expression ) [ PERCENT ] ]
    [ FROM ]
        { table_name [ WITH ( <table_hint_limited> [ ...n ] ) ]
        | view_name
        | rowset_function_limited
        | table_valued_function
    }
    [ <OUTPUT Clause> ]
    [ FROM <table_source> [ ,...n ] ]
    [ WHERE { <search_condition>
            | { [ CURRENT OF
                  { { [ GLOBAL ] cursor_name }
                      | cursor_variable_name
                  }
                ]
              }
            }
    ]
```

```
    [ OPTION ( <Query Hint> [ ,...n ] ) ]
[; ]

<object> ::=
{
    [ server_name.database_name.schema_name.
      | database_name. [ schema_name ] .
      | schema_name.
    ]
        table_or_view_name
}
```

Os argumentos mais importantes da cláusula **DELETE** estão detalhados abaixo:

- TOP expression - Especifica o número ou percentagem de tuplos aleatórios que serão inseridos.
- FROM - Palavra opcional entre a instrução DELETE e o nome da tabela alvo.
- server_name - Nome do servidor onde a tabela está guardada.
- database_name - Nome da base de dados que contém a tabela.
- table_name - Nome da tabela onde serão removidos tuplos.
- WHERE <search_condition> - Restringir os tuplos a serem removidos. Restringir segundo as condições impostas na clausula WHERE.
- WITH (<table_hint_limited> [... n]) - Especifica as opções que podemos permitir para a tabela de destino. Um exemplo disso, é especificar qual o recurso bloqueado aquando da inserção. (p.e: TABLOCK). A sintaxe para esta opção é a seguinte:

```
<table_hint_limited> ::=
{
    KEEPIDENTITY
  | KEEPDEFAULTS
  | FASTFIRSTROW
  | HOLDLOCK
  | IGNORE_CONSTRAINTS
  | IGNORE_TRIGGERS
  | NOWAIT
    | PAGLOCK
  | READCOMMITTED
  | READCOMMITTEDLOCK
  | READPAST
  | REPEATABLEREAD
  | ROWLOCK
  | SERIALIZABLE
  | TABLOCK
```

```
    | TABLOCKX
    | UPDLOCK
    | XLOCK
}
```

A cláusula **FROM** é utilizada em todas as anteriores cláusulas, portanto a sua definição é a seguinte:

```
[ FROM { <table_source> } [ ,...n ] ]
<table_source> ::=
{
        table_or_view_name [ [ AS ] table_alias ] [ <
            tablesample_clause> ]
        [ WITH ( < table_hint > [ [ , ]...n ] ) ]
    | rowset_function [ [ AS ] table_alias ]
        [ ( bulk_column_alias [ ,...n ] ) ]
        | user_defined_function [ [ AS ] table_alias ] [ (
            column_alias [ ,...n ] ) ]
    | OPENXML <openxml_clause>
    | derived_table [ AS ] table_alias [ ( column_alias [ ,...n
      ] ) ]
    | <joined_table>
    | <pivoted_table>
    | <unpivoted_table>
      | @variable [ [ AS ] table_alias ]
      | @variable.function_call ( expression [ ,...n ] ) [ [
          AS ] table_alias ] [ (column_alias [ ,...n ] ) ]


    <joined_table> ::=
  {
      <table_source> <join_type> <table_source> ON <
          search_condition>
      | <table_source> CROSS JOIN <table_source>
      | left_table_source { CROSS | OUTER } APPLY
          right_table_source
      | [ ( ] <joined_table> [ ) ]
  }
<join_type> ::=
    [ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } [ <
        join_hint> ] ]
    JOIN
}
```

Os argumentos mais importantes da cláusula **FROM** estão detalhados abaixo:

- <table_source> - Tabela usada na instrução T-SQL.
- table_or_view_name - Indica o nome da tabela ou da vista, usada para a instrução T-SQL.

- WITH (<table_hint>) - Especifica que o query optimizer usa uma optimização ou uma estratégia de locks para a tabela referida. As estratégias de Lock permitidas são: NOLOCK, READUNCOMMITTED, UPDLOCK, REPEATABLEREAD, SERIALIZABLE, READCOMMITTED, FASTFIRSTROW, TABLOCK, TABLOCKX, PAGLOCK, ROWLOCK, NOWAIT, READPAST, XLOCK, and NOEXPAND. A estratégia de locks é importante para transacções.

- <join_type> Especifica qual o tipo de junção a ser usado numa junção (se houver junção).

2.3 Tipos

O SQL Server fornece muitas opções para criação e armazenamento de dados. Estes dados terão que ter um tipo associado. O SQL Server permite vários tipos de dados, e esses tipos estão divididos por categorias.

Categorias

Numéricos Exactos	Strings unicode
Numéricos Aproximados	Caracteres Binários
Data e Hora	Outros tipos de dados
Strings	

Cada categoria tem vários tipos de dados. Abaixo estão listados os tipos de dados associados a cada categoria.

2.3.1 Numéricos Exactos

Tipo	**Descriçao**
int	Inteiro de 32 bit
bigint	Inteiro de 64 bits
smallint	Inteiro de 16 bits
tinyint	Inteiro de 8 bits

De referir que o SQL Server não tem um tipo booleano. Portanto esse valor booleano é simulado por um tipo inteiro, designado por **Bit**. Se o valor for 1, então o valor booleano associado é True, e se o valor for 0, o valor booleano associado é False.

Numéricos Aproximados

Tipo	**Tamanho**	**Precisão**
float	32 Bits (4 bytes)	7 Dígitos
float	64 Bits (8 bytes)	15 Dígitos
real	32 Bits (4 bytes)	7 Dígitos

2.3.2 Data e Hora

- date
- datetimeoffset
- datetime2
- smalldatetime
- datetime
- time

2.3.3 Strings Unicode

- nchar
- nvarchar
- ntext

2.3.4 Strings Binárias

- Binary
- Image
- varbinary

2.3.5 Strings

- char
- varchar
- text

2.3.6 Outros tipos de dados

- cursor
- timestamp
- hierarchyid
- uniqueidentifier
- sql_variant
- xml
- table

Tabela geral de tipos de dados

bigint	binary	bit	char	CLR
cursor	date	datetime	datetime2	datetimeoffset
decimal	float	hierarchyid	image	int
money	nchar	ntext	numeric	nvarchar
real	rowversion	smalldatatime	smallint	smallmoney
sql_variant	table	text	time	timestamp
int, bigint, smallint, and tinyint	varbinary	varchar	uniqueidentifier	xml

Capítulo 3

Armazenamento e file structure

Nesta secção iremos debater toda a estrutura de armazenamento de dados que o SQL Server possui de forma a gerir a informação que irá estar presente dentro de qualquer base de dados implementada. Também aprofundaremos as noções de como esses dados são guardados em ficheiros do sistema de forma a permitir uma fácil gestão destes.

Estes tópicos serão explorados ao detalhe, sendo especificadas todas as suas implementações possíveis no SQL Server, explicitando ainda a maneira como funcionam de forma a poder-se compreender melhor as bases de armazenamento de dados deste sistema de bases de dados.

Inicialmente irá ser falada a gestão da memória e o buffer de gestão de dados, com as suas políticas de substituição. De seguida serão introduzidas as páginas do sistema que guardam os dados, bem como as extensões e como elas são guardadas dentro de ficheiros. Seguidamente falar-se-á das estruturas de dados que suportam toda a informação das bases de dados como partições, árvores B+, heap's, etc. Para finalizar será feita uma breve apresentação das políticas de recuperação de dados e backup destes dentro do sistema do SQL Server.

3.1 Gestão de memória

Vamos iniciar o estudo do armazenamento e estrutura de ficheiros do SQL Server pela arquitectura de gestão de memória explicando como funciona e a sua importância para o bom desempenho das bases de dados em implementadas no SQL Server.

O SQL Server adquire ou liberta espaço em memória segundo o que necessita no momento não tendo o administrador de especificar quanta memória quer disponível, sendo no entanto possível este especificar tal atributo. Como todas as aplicações de 32 bits têm 4 GB de espaço de endereçamento, sendo utilizados 2GB pelo kernel do Windows, cada aplicação poderá ter 2GB deste espaço. No entanto o SQL Server dispõe de um sistema de Adress Windowing

Extension (AWE), que permite à aplicação dispor de tanta memória quanta o sistema operativo tiver. Desta feita o SQL Server poderá correr com um espaço de endereçamento até 64 GB. O SQL Server também usa o bloqueio de páginas em memória para impedir o sistema operativo de utilizar esse espaço.

A informação da base de dados está guardada em ficheiros que estão em disco e como a leitura destes é muito mais lenta que a leitura da memória. Como um dos principais objectivos do SQL Server é minimizar o I/O do disco, este constrói um buffer de cache em memória para guardar páginas lidas da base de dados. O SQL Server tem uma grande preocupação em diminuir as leituras e escritas físicas do disco por gastarem muitos recursos, tendo dois objectivos principais: impedir que o buffer cresça tanto que o sistema fique sem memória e minimizar o I/O físico da base de dados maximizando o tamanho do buffer de cache.

Para tornar estes objectivos possíveis o SQL Server tem um sistema de gestão do buffer referido anteriormente, sendo fulcral para ter a eficácia nos objectivos traçados. A gestão do buffer consiste em dois mecanismos: o gestor do buffer para aceder e actualizar páginas e o buffer de cache para reduzir o I/O com os ficheiros da base de dados.

O buffer de gestão é uma página de 8 KB em memória do tamanho dos dados (como veremos mais à frente), dividindo assim o buffer de cache em várias páginas de 8 KB. O gestor do buffer controla as funções de leitura de dados ou páginas de índex dos ficheiros de disco da base de dados para o buffer de cache e as escritas das páginas modificadas para disco. Uma página fica no buffer de cache até que o gestor precise de espaço para ler mais dados, sendo que informação só é escrita em disco se foi modificada uma ou várias vezes quando estava no buffer. A esta política de substituição no buffer chama-se clock replacement policy. As leituras e escritas feitas por este componente são assíncronas.

Ao iniciar o SQL Server reserva o espaço para o buffer de cache segundo alguns parâmetros como a memória física do sistema e o número máximo de threads do servidor. No entanto este só preenche inicialmente a carga corrente da memória física actual. Durante este processo até o SQL Server obter a memória que necessita para o buffer, pedidos de leitura vão enchendo o buffer quando necessário. Este tempo chama-se ramp-up e é especializado em tornar leituras de uma página em pedidos de oito páginas tornando mais rápida a inicialização.

O buffer utiliza quase toda a memória disponível para o SQL Server interagindo com o gestor de recursos, o sistema operativo do SQL Server e o gestor de logs de maneira a maximizar a sua eficiência dentro do sistema. O gestor do buffer suporta ainda algumas funcionalidades de forma a melhorar o seu funcionamento, estas são: o buffer está preparado para acessos de dados não uniformes, suporta Hot Add Memory permitindo a adição de memória física sem recomeçar o servidor e permite gerir memória dinamicamente.

Esta gestão dinâmica é feita através de mecanismos que ajustam a memória necessária tendo em conta o congestionamento do serviço e limites superiores e inferiores da memória do sistema.

Quando outras aplicações correm na máquina, diminuindo o espaço em memória dedicado para o SQL Server, este gestor consegue ajustar rapidamente o consumo de memória da base de dados aumentando ou diminuindo o espaço para o SQL Server. Estes ajustes são rápidos tendo em conta a situação geral da máquina que está a correr a base de dados.

Para gerir as grandes bases de dados o SQL Server utiliza o seu componente de AWE já referido de maneira a poder guardar grandes quantidades de dados em memória evitando muitos acessos ao disco, de forma a aumentar a performance nestas bases de dados de grandes dimensões que poderiam trazer pouca eficiência no que toca ao SQL Server.

3.2 Ficheiros e páginas

Depois de explicada a arquitectura de memória do SQL Server com o seu sistema de gestão do buffer e o buffer de cache que são implementados de forma a melhorar os acessos aos ficheiros no disco da base de dados, vamos mostrar como os dados são guardados nestes ficheiros e todas as construções envolventes na estruturação da informação.

O SQL Server mapeia a base de dados num conjunto de ficheiros do sistema operativo. Informação e logs são sempre guardados em ficheiros diferentes, sendo que cada ficheiro é só utilizado para uma base de dados enquanto que os filegroups são vários ficheiros que coordenam a localização dos dados e tarefas de recuperação e backup.

No SQL Server existem três tipos de ficheiros para guardar dados relativos às bases de dados. Os ficheiros primários são os mais importantes dentro de uma base de dados e têm apontadores para outros ficheiros dentro da base de dados sendo a sua extensão .mdf. Os ficheiros secundários organizam os dados e uma base de dados pode não ter ficheiros deste tipo dependendo das suas características, sendo também possível ter vários. Estes ficheiros que gerem os metadados da aplicação servem como dicionários de dados. A sua extensão é .ndf. Os ficheiros de log guardam toda a informação de logs que é utilizada para recuperar a base de dados sendo necessário pelo menos um numa. A extensão é .ldf. Estas extensões não são obrigatórias, mas sim recomendadas.

No SQL Server a localização de todos os ficheiros numa base de dados estão guardados no ficheiro primário ou na base de dados master. Esta é uma base de dados que guarda toda a informação de um sistema SQL Server. Dados sobre informações dentro do sistema (contas, configuração, servidores) são guardados nesta base de dados, bem como registos de todas as outras bases de dados e informações de inicialização sendo que desta maneira o SQL

Server nunca inicia se a base de dados master não estiver operacional. Os objectos do sistema são guardados numa base de dados de recursos, separada desta. O SQL Server utiliza quase sempre o ficheiro de localização desta base de dados, excepto em algumas excepções onde utiliza o ficheiro primário.

Os ficheiros têm dois tipos de nomes: os lógicos e os os (sistema operativo). Os lógicos servem para guardar informações de uma frase de Transact-SQL. Os nomes de ficheiros de sistema operativo são o nome do ficheiro físico e a sua directoria seguindo as regras do sistema operativo corrente. Desta maneira os ficheiros de dados e log do SQL Server podem ser guardados tanto em FAT como NTFS, sendo recomendado o NTFS por questões de segurança. Existem algumas limitações com o sistema de compressão de ficheiros do NTFS quanto a ficheiros de log e filegroups. Quando há várias instancias do SQL Server a correr na mesma máquina, cada um tem uma directoria diferente por defeito para guardar os ficheiros dessas bases de dados.

Uma base de dados de SQL Server pode guardar no máximo 2^{31} objectos e ter vários ficheiros de sistema operativo com um tamanho máximo de 2^{20} TB.

Páginas de ficheiros de dados são numeradas sequencialmente começando no 0 para a primeira página de determinado ficheiro. Cada ficheiro numa base de dados tem um identificador numérico único que o diferencia dos outros, sendo necessário para identificar uma página numa base de dados, o identificador desta e do respectivo ficheiro onde se localiza.

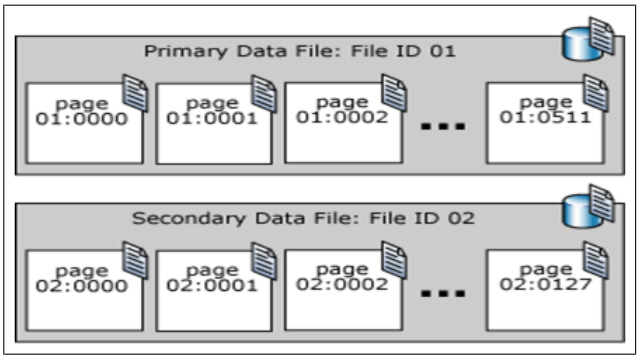


Figura 3.1: *Organização de páginas em ficheiros*

Exemplo de um ficheiro da numeração e identificação de páginas num ficheiro primário de 4 MB e um secundário de 1 MB. A primeira página de cada ficheiro é um cabeçalho que guarda informações sobre os atributos do ficheiro e muitas outras páginas no início do ficheiro também podem conter outras informações como alocação de mapas. Existe uma página de boot contendo informação sobre o sistema que é guardada tanto nos ficheiros de dados como nos de log.

A página é a unidade de informação do SQL Server e compondo estas os

ficheiros, as operações de leitura e escrita são feitas a páginas. Uma extensão é uma lista de 8 páginas, sendo que todas as páginas estão guardadas em extensões de maneira a facilitar a gestão destas. No SQL Server uma página tem 8 KB havendo portanto 128 páginas por cada MB. Cada página tem os cabeçalhos já discutidos de 96 bytes incluindo informação de identificadores.

Existem vários tipos de página para guardar determinados dados: página de dados, de índex, de texto ou imagem e informações relevantes já discutidas. Linhas de dados são colocadas na página em série começando depois do cabeçalho. O offset das linhas começa no fim da página e cada uma delas tem uma tabela que referência a cada linha da página. Estas entradas estão em ordem inversa das linhas da página sendo que cada entrada guarda o quão longe do inicio da página está o primeiro byte dessa linha.

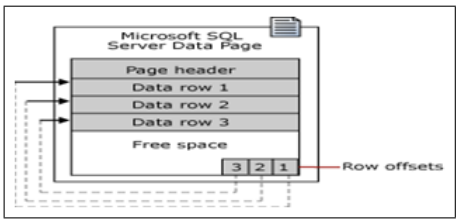


Figura 3.2: *Organização de páginas*

Se uma linha guardar mais 8 KB de memória, o SQL Server gere esse facto dinamicamente movendo colunas de dados overflow para páginas de forma a guardar um apontador para a página inicial, guardando os dados. Esta gestão é feita quer as linhas aumentem ou diminuam de tamanho.

3.3 Extensões e filegroups

Extensões são a unidade básica de espaço gerida pelo sistema sendo o seu tamanho de 64 KB. Para que a gestão do espaço seja eficiente o SQL Server não guarda extensões para tabelas com poucos dados. Desta forma as páginas de cada extensão podem ser de objectos diferentes, referenciando tabelas diferentes.

Uma nova tabela ou índice é normalmente armazenada numa extensão mista (referencia várias tabelas) e à medida que vai crescendo essa extensão pode ser uniformizada, se já tiver tamanho suficiente na sua criação a extensão é logo uniformizada.

Para gerir onde se guardam as extensões e o espaço livre de forma a diminuir o número de acessos a disco e número de vezes que as páginas têm que ser reorganizadas o SQL Server tem duas estruturas de dados: o GAM e o SGAM. Cada um tem um bit para indicar se está a ser utilizado ou não essa extensão (GAM) ou se é uma extensão mista ou não (SGAM). Existe também o PFS para detectar o espaço livre e marca-lo sendo mantido quanto

Figura 3.3: *Tipos de extensões*

espaço está livre para estruturas de heap e páginas de texto ou imagem.

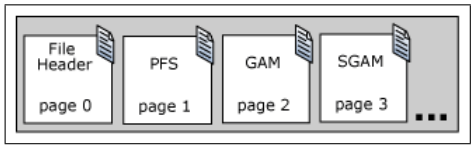


Figura 3.4: *Gestão de extensões*

Existem também estruturas para gerir o espaço gasto por objectos nas tabelas e para marcar quais as extensões que foram modificadas, aumentando em muito a eficiência do sistema.

O tamanho dos ficheiros no SQL Server cresce automaticamente a partir do seu tamanho original, crescendo este por um determinado incremento que pode ser escolhido. Os filegroups só podem crescer quando todos os seus ficheiros estão cheios. Limites de tamanho podem ou não ser escolhidos, sendo que sem limite um ficheiro cresce até não haver espaço.

Por motivos de administração, tanto ficheiros da base de dados como objectos podem ser juntos em filegroups. Existem os filegroups primários que contêm o ficheiro de dados primário e todas as páginas das tabelas do sistema. Os definidos pelo utilizador definem-se com o "FILEGROUP"ao criar ou alterar tabelas. É definido um filegroup por defeito onde as páginas serão localizadas podendo este ser mudado apenas por membror de db_owner. Nenhum ficheiro pode estar em mais que um filegroup, sendo que tabelas índices ou grandes objectos de dados podem ser adicionados a um. Desta maneira todas as suas páginas serão adicionadas ao filegroup ou as tabelas e os índices podem ser particionados podendo ficar dados de tabelas ou índices em vários filegroups aquando deste particionamento. Esta estrutura está resumida na imagem seguinte.

De seguida iremos descrever como é que as tabelas são organizadas guardadas e acedidas dentro das extensões.

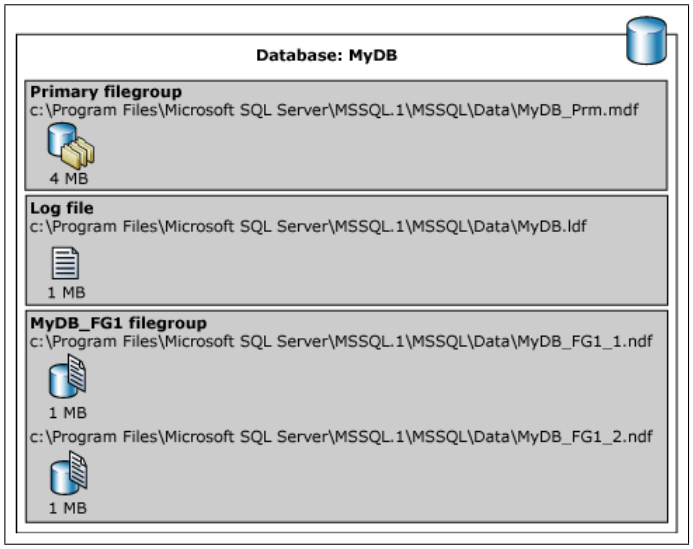


Figura 3.5: *Exemplo da organização de uma BD*

3.4 Estrutura de dados e partições

Uma tabela está contida numa ou mais partições e cada partição contém linhas de dados num heap ou numa estrutura de índices agrupados (clustered). Estes são geridos numa ou mais unidades de localização dependendo dos tipos de colunas nas linhas de dados.

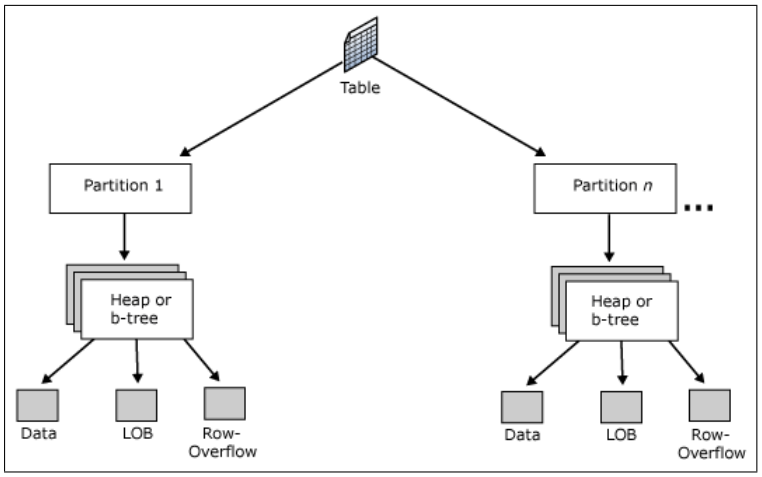


Figura 3.6: *Arquitectura de uma tabela*

Uma partição é uma unidade de organização dados definida pelo utilizador e por defeito cada tabela ou índice contém uma partição que tem todas as suas páginas estando essa partição sempre só num filegroup. Quando uma tabela ou índice utiliza múltiplas partições os dados são divididos hor-

izontalmente para que os grupos de linhas sejam mapeados em partições individuais baseadas numa coluna específica. Essas partições podem ser colocadas em mais que um filegroup na base de dados. As tabelas ou índices são tratados como uma entidade lógica única quando os dados são acedidos.

Este particionamento torna as tabelas ou índices mais fáceis de gerir pois os vários conjuntos de dados podem ser geridos e acedidos rapidamente e com eficiência. Desta forma é mais rápida a manutenção de certos tipos de dados bem como ir buscar esses mesmos dados, sendo as diferenças de tempo consideráveis.

Para criar uma partição utiliza-se o seguinte comando[1]:

```
CREATE PARTITION FUNCTION partition_function_name (
    input_parameter_type )
AS RANGE [ LEFT | RIGHT ]
FOR VALUES ( [ boundary_value [ ,...n ] ] )
[ ; ]
```

Existem duas maneiras distintas para se organizarem as páginas de tabelas no SQL Server dentro de uma partição. Tabelas clustered são tabelas que têm um índice clustered estando as linhas de dados organizadas segundo a chave do índice. As páginas em cada nível do índice incluindo as páginas de dados no nível da folha ligadas por uma lista duplamente ligada. A navegação feita entre níveis é determinada por chaves. Esta é a estrutura de uma árvore B+ com índices. Estes clusters de informação são criados como índices normais (discutido noutro capitulo).

Para tabelas sem índice clustered são usados heaps. Nestes heaps as linhas de dados não estão organizados em nenhuma ordem específica bem como as páginas não estão ordenadas e não sendo guardadas numa lista duplamente ligada. Um heap por defeito está só numa partição, mas se um heap tem várias partições, cada partição tem uma estrutura heap que guarda os seus dados. O SQL Server utiliza páginas IAM para percorrer o heap, sendo que estas representam extensões na mesma ordem que elas aparecem no ficheiro. Desta maneira para procura numa tabela, apenas são pesquisadas as páginas IAM segundo a imagem seguinte. De notar que os dados podem não aparecer na ordem em que foram inseridos mas que o ficheiro é pesquisado sequencialmente através das páginas IAM.

[1] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

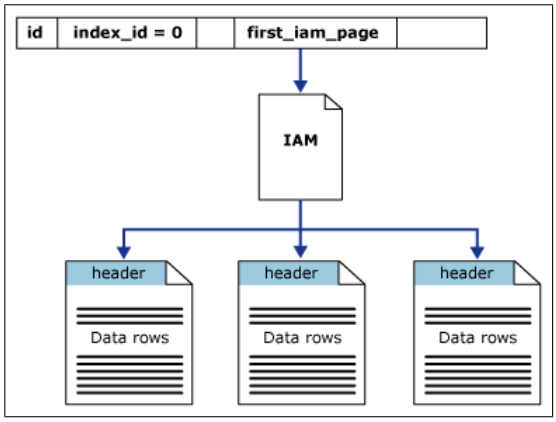


Figura 3.7: *Páginas IAM em heaps*

Índices não clustered organizam-se em árvores B+ como os que falamos anteriormente. A diferença entre estes é que estes não afectam a ordem das linhas de dados sendo que o nível de folha contem índices que contêm a chave e o valor. A imagem a seguir servirá para explicitar melhor esta organização.

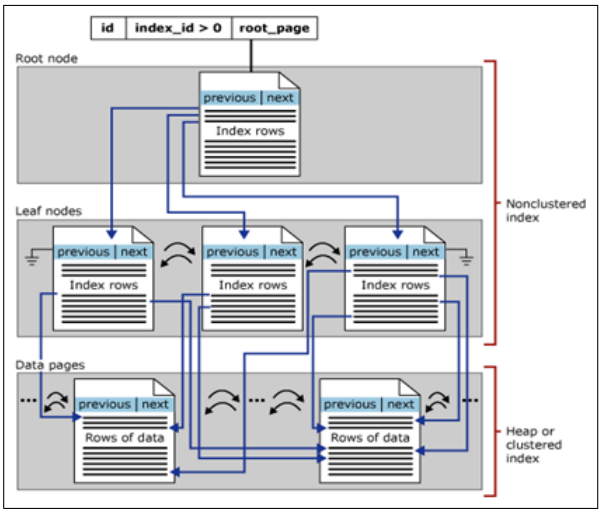


Figura 3.8: *Índices não clustered*

No SQL Server também pode ser utilizado XML para guardar algumas informações que pretendermos. Esta é uma opção a considerar quando as tabelas têm uma estrutura muito fixe e propensa ao XML. Desta forma poder-se-á tornar a base de dados mais forte tendo em conta as qualidades do XML sempre que as tabelas tenham uma estrutura em que se aplique este caso.

Unidades de alocação são páginas dentro de um heap ou uma árvore B+ usadas para gerir dados baseados no seu tipo. Existem três tipos de unidades diferentes, sendo que numa partição específica uma estrutura qualquer só

pode ter um dos tipos. Existe um tipo para dados normais, um para objectos muito grandes e outro de overflow para dados de tamanho variável.

3.5 Recuperação de dados

Para finalizar falaremos um pouco dos mecanismos que o SQL Server tem para recuperar os dados após uma queda da base de dados. Este tem vários modos de recuperação. Um modo simples onde só é recuperada a última base de dados em backup. Um completo em que guarda ficheiros de log em backup e não é perdido nenhum trabalho normalmente e outro idêntico ao completo mas que permite uma performance elevada em certos casos.

Todos estes modos de recuperar informação têm por base métodos de backup das bases de dados. Existem também vários modos de backup desde backups parciais a totais, dependendo das características do sistema. Com estes dois mecanismos a funcionar em conjunto, o SQL Server permite a longevidade dos seus dados mesmo após situações anormais.

Capítulo 4

Indexação

O objectivo da Indexação é de melhorar o acesso aos dados. Podemos indexar um ou mais campos, de forma a que as querys corram da melhor forma possível. Mas se o utilizador criar mal os seu indexes, podemos ter uma reacção contraria, a de atrasar as querys em vez de melhorar.

4.1 Sintaxe

4.1.1 Criação de Índices Relacionais[11]

Um índice em Sql Server tem a seguinte sintaxe[1]:

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
    [ FILESTREAM_ON { filestream_filegroup_name |
      partition_scheme_name | "NULL" } ]

[ ; ]
```

O índice é criado com o nome *index_name* sobre uma tabela ou view *<object>*. Os campos onde vão ser aplicados o índice são os com o nome *column* e com a ordem *ASC* ou *DESC*.

Podemos criar índices com 16 colunas e o tamanho dos valores dos campos não pode ultrapassar os 900 bytes. Os campos com tipo de dados *large object* (LOB), como por exemplo, *ntext*, *text*, *varchar(max)*, *nvarchar(max)*,

[1] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

varbinary(max), *xml* e *image* não podem ser chaves de índices. Se indicarmos que este é *UNIQUE* então não permite que duas linhas tenha os mesmos valores nos campos do índice.

A ordem pré-definia dos campos é Ascendente(*ASC*). Poderemos indicar se é *CLUSTERED* ou *NONCLUSTERED*.

Podemos indicar campos não chave adicionais para serem adicionados no índice dos *NONCLUSTERED*. Para adicionar estes campos usamos a a palavra reservada *INCLUDE* e a lista dos campos.

A clausula *WHERE* cria um índice filtrado, ou seja, um índice com algumas linhas do objecto <*object*>.

A clausula *WITH* serve para indicar-mos opções da criação do índice. As opções são as seguintes:

- **PAD_INDEX = { ON | OFF }**: Especifica o *padding* do índice. Se estiver *ON*, a percentagem de espaço livre que é especificado por o *fillfactor* é aplicado as páginas intermédias do índice. Se estiver *OFF* ou o *fillfactor* não for especificado, as páginas intermédias do índice são cheias.

- **FILLFACTOR = fillfactor**: Especifica a percentagem de preenchimento das folhas durante a construção do índice ou no *rebuild*. A percentagem toma valores entre 0 e 100. Se tomar os valores 0 ou 100, cria um índice com as folhas cheias.

- **SORT_IN_TEMPDB = { ON | OFF }**: Especifica se ordena os resultados numa tabela temporária.

- **IGNORE_DUP_KEY = { ON | OFF }**: Especifica o tipo de erro é gerado quando se introduz valores repetidos num índice único. Se estiver *ON*, lançará um *warning* e as linhas que violem as restrições não serão inseridas. Se estiver *OFF*, lançará um erro e todo o *insert* não será inserido.

- **STATISTICS_NORECOMPUTE = { ON | OFF }**: Especifica se as estatísticas são recalculadas. Se estiver *OFF*, estatísticas desactualizadas não são recalculadas automaticamente, se estiver *ON* são calculadas automaticamente.

- **DROP_EXISTING = { ON | OFF }**: Especificas se existir um outro índice com o mesmo nome é apagado ou não. Se estiver *ON* então o índice é apagado e é criado o novo, se estiver *OFF* então é gerado um erro.

- **ONLINE = { ON | OFF }**: Especifica se as tabelas e os seus índices estão disponíveis para consultas e modificação durante a operação de índice. Esta opção só se encontra disponível nas versões SQL Server Enterprise, Developer, e de Evaluation.

- **ALLOW_ROW_LOCKS = { ON | OFF }**: Especifica se os bloqueios às linha são permitidos.
- **ALLOW_PAGE_LOCKS = { ON | OFF }**: Especifica se os bloqueios a páginas são permitidos.
- **MAXDOP = max_degree_of_parallelism**: Especifica o número de processadores a serem usados no plano de execução paralela. O número máximo de processadores é 64.

A clausula "*ON* partition_scheme_name (column_name)"serve para especificarmos o tipo de partições do índice.

A clausula "*ON* filegroup_name"cria o índice no especificado *filegroup*.

A clausula *FILESTREAM_ ON* especifica a colocação dos dados *FILESTREAM* para a tabela quando o índice é criado. O *FILESTREAM_ ON* permite os dados *FILESTREAM* movidos para diferentes *FILESTREAM filegroup* ou esquema de partição.

4.1.2 Criação de Índices XML[13][21]

O índices de XML podem ser criados sobre campos com tipos de dados XML. Este indexa todas as *tags*, valores e caminhos sobre a instância XML na coluna e beneficia do desempenho da coluna.

As aplicações podem beneficiar dos índices de XML nas seguintes situações:

- Se as queries sobre colunas de XML forem comuns. O custo de manutenção dos índices XML durante alteração dos dados têm de ser considerados
- Os dados de XML forem bastante grandes e os bocados pretendidos são pequenos. Construir índices de XML evita o *parse* de todos os dados em *run time* e beneficia das pesquisas por índices para o processamento de queries eficiente.

Existem duas categorias neste tipo de índices:

- *Primary XML index*
- *Secondary XML index*

O primeiro índice num campo do tipo de dados XML tem de ser *Primary XML index*. Usando um *Primary XML index*, os seguintes tipos de *Secondary XML index* são suportados: caminho, valor e propriedades. Dependendo do tipo de *queries* os *Secondary XML index* podem melhorar o desempenho da *query*.

Quando indicamos que um índice é *PRIMARY* então um índice *clustered* é criado com a chave formada a partir de a chave da tabela e o identificador do nó XML. Cada tabela pode ter até 249 índices de XML. Quando criar um índice de XML tem de ter em conta:

- Um indice *clustered* tem de existir como *primary key* da tabela
- A chave da *primary key* tem de ser limitada a 15 campos
- Cada campo de XML pode ter um *primary XML index* e múltiplos *secondary XML indexes*
- Um *primary XML index* num campo de XML tem de existir antes de um *secondary XML index*
- Um índice de XML só pode ser criado sobre um campo XML. Não pode criar índices de XML em campos que não são XML e não pode criar um índice normal em campos de XML
- Não pode criar índices XML em campos XML em *views*

A sintaxe[2] para criar um índice de XML é:

```
CREATE [ PRIMARY ] XML INDEX index_name
    ON <object> ( xml_column_name )
    [ USING XML INDEX xml_index_name
        [ FOR { VALUE | PATH | PROPERTY } ] ]
    [ WITH ( <xml_index_option> [ ,...n ] ) ]
[ ; ]
```

O índice é criado com o nome *index_name* na tabela *<object>* sobre o campo *xml_column_name*. Se indicarmos a opção *USING XML INDEX xml_index_name* então estamos a dizer que o seguinte índice secundário XML vai usar a chave do índice de XML primário com o nome *xml_index_name* como combinação da usa chave. Para indicarmos que tipo de dados vamos indexar usamos a opção *FOR { VALUE | PATH | PROPERTY }*.

A clausula *WITH* serve para indicar-mos opções da criação do índice. As opções são as seguintes:

- **PAD_INDEX = { ON | OFF }**: Especifica o *padding* do índice. Se estiver *ON*, a percentagem de espaço livre que é especificado por o *fillfactor* é aplicado as páginas intermédias do índice. Se estiver *OFF* ou o *fillfactor* não for especificado, as páginas intermédias do índice são cheias.

- **FILLFACTOR = fillfactor**: Especifica a percentagem de preenchimento das folhas durante a construção do índice ou no *rebuild*. A percentagem toma valores entre 0 e 100. Se tomar os valores 0 ou 100, cria um índice com as folhas cheias.

- **SORT_IN_TEMPDB = { ON | OFF }**: Especifica se ordena os resultados numa tabela temporária.

- **IGNORE_DUP_KEY = { ON | OFF }**: Especifica o tipo de erro é gerado quando se introduz valores repetidos num índice único. Se estiver *ON*, lançará um *warning* e as linhas que violem as restrições não serão inseridas. Se estiver *OFF*, lançará um erro e todo o *insert* não será inserido.

- **STATISTICS_NORECOMPUTE = { ON | OFF }**: Especifica se as estatísticas são recalculadas. Se estiver *OFF*, estatísticas desactualizadas não são recalculadas automaticamente, se estiver *ON* são calculadas automaticamente.

- **DROP_EXISTING = { ON | OFF }**: Especificas se existir um outro índice com o mesmo nome é apagado ou não. Se estiver *ON*

[2] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

então o índice é apagado e é criado o novo, se estiver *OFF* então é gerado um erro.

- **ALLOW_ROW_LOCKS = { ON | OFF }**: Especifica se os bloqueios às linha são permitidos.
- **ALLOW_PAGE_LOCKS = { ON | OFF }**: Especifica se os bloqueios a páginas são permitidos.
- **MAXDOP = max_degree_of_parallelism**: Especifica o número de processadores a serem usados no plano de execução paralela. O número máximo de processadores é 64.

4.1.3 Criação de Índices *SPATIAL*[12][32]

O SQL Server 2008 suporta dados espaciais. Isto inclui suporte para planos geométricos, os seja, pontos, linhas e poligonos num sistema de coordenadas euclidianas. O tipo de dados *geography* representa objectos geográficos na superfície da Terra, como um terreno. Um Índice *SPATIAL* num campo *geography* mapeia dados geográficos para um espaço bidimensional não-euclidiano.

Um índice *SPATIAL* é definido num campo de uma tabela que contêm dados geográficos. Cada índice *SPATIAL* refere-se a um espaço finito.

A sintaxe[3] para criar um índice *SPATIAL* é:

```
CREATE SPATIAL INDEX index_name
  ON <object> ( spatial_column_name )
    {
       [ USING <geometry_grid_tessellation> ]
          WITH ( <bounding_box>
                 [ [,] <tesselation_parameters> [ ,...n ] ]
                 [ [,] <spatial_index_option> [ ,...n ] ] )
     | [ USING <geography_grid_tessellation> ]
          [ WITH ( [ <tesselation_parameters> [ ,...n ] ]
                   [ [,] <spatial_index_option> [ ,...n ] ] ) ]
    }
  [ ON { filegroup_name | "default" } ]
;
```

O índice é criado com o nome *index_name* na tabela <*object*> sobre o campo *spatial_column_name*.

A clausula *USING* indica o esquema de mosaico para o índice *SPATIAL*. Esta clausula pode tomar os seguintes valores:

- *geometry*
- *geography*

Um indice *SPATIAL* só pode ser criado sobre campos do tipo *geometry* ou *geography*, caso contrario, o SQL Server indicará um erro.

A clausula "*ON* filegroup_name"cria o índice no especificado *filegroup*.

A clausula *WITH* serve para indicar-mos opções da criação do índice. As opções são as seguintes:

- **GEOMETRY_GRID**: Especifica o *grid tessellation* em *geometry*
- **GEOGRAPHY_GRID**: Especifica o *grid tessellation* em *geography*
- **BOUNDING_BOX**: Especifica um quadruplo numérico que define quatro coordenadas de uma caixa delimitadora. O *x-min* e *y-min* do canto inferior esquerdo e *x-max*, *y-max* do canto superior direito

[3] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

- GRIDS: Define a densidade da *grid* em cada nível. Os níveis possíveis são:
  - LEVEL_1
  - LEVEL_2
  - LEVEL_3
  - LEVEL_4

  E as densidades possíveis são:
  - LOW
  - MEDIUM
  - HIGH
- **CELLS_PER_OBJECT = n**: Especifica o numero de células *tessellation* por um único objecto *spatial* num índice por um processo *tessellation*. N pode ser um inteiro entre 1 e 8192, inclusive.
- **PAD_INDEX = { ON | OFF }**: Especifica o *padding* do índice. Se estiver *ON*, a percentagem de espaço livre que é especificado por o *fillfactor* é aplicado as páginas intermédias do índice. Se estiver *OFF* ou o *fillfactor* não for especificado, as páginas intermédias do índice são cheias.
- **FILLFACTOR = fillfactor**: Especifica a percentagem de preenchimento das folhas durante a construção do índice ou no *rebuild*. A percentagem toma valores entre 0 e 100. Se tomar os valores 0 ou 100, cria um índice com as folhas cheias.
- **SORT_IN_TEMPDB = { ON | OFF }**: Especifica se ordena os resultados numa tabela temporária.
- **STATISTICS_NORECOMPUTE = { ON | OFF }**: Especifica se as estatísticas são recalculadas. Se estiver *OFF*, estatísticas desactualizadas não são recalculadas automaticamente, se estiver *ON* são calculadas automaticamente.
- **DROP_EXISTING = { ON | OFF }**: Especificas se existir um outro índice com o mesmo nome é apagado ou não. Se estiver *ON* então o índice é apagado e é criado o novo, se estiver *OFF* então é gerado um erro.
- **ALLOW_ROW_LOCKS = { ON | OFF }**: Especifica se os bloqueios às linha são permitidos.
- **ALLOW_PAGE_LOCKS = { ON | OFF }**: Especifica se os bloqueios a páginas são permitidos.

- **MAXDOP = max_degree_of_parallelism**: Especifica o número de processadores a serem usados no plano de execução paralela. O número máximo de processadores é 64.

4.1.4 Criação de Índices *FULLTEXT*[10][5][18][33]

As *queries FULL-TEXT* realizam pesquisas linguísticas sobre os dados de texto nos índices *FULL-TEXT*, operando em palavras e fases com base em regras linguísticas de um determinado idioma.

O SQL Server 2008 disponibiliza a funcionalidade executar *queries FULL-TEXT* em vez de se usar as *queries* baseadas por caracteres. Antes de as *queries FULL-TEXT* poderem usadas numa tabela, o administrador da base de dados tem de criar um índice *FULL-TEXT* na tabela. O índice *FULL-TEXT* pode ser criado sobre um ou mais campos de texto. Estes campos podem ser destes tipos de dados:

- *char*
- *varchar*
- *nchar*
- *nvarchar*
- *text*
- *ntext*
- *image*
- *xml*
- *varbinary*
- *varbinary(max)*

Cada índice *FULL-TEXT* indexa uma ou mais colunas e cada coluna pode ter uma idioma definida. No SQL Server 2008, os índices de *FULL-TEXT* suporta mais de 50 idiomas diferentes, como o Inglês, Português, Espanhol, Chinês, Japonês, Árabe, Bengali e Hindu.

A sintaxe[4] para criar um índice *FULLTEXT* é:

```
CREATE FULLTEXT INDEX ON table_name
      [ ( { column_name
              [ TYPE COLUMN type_column_name ]
              [ LANGUAGE language_term ]
          } [ ,...n]
            ) ]
    KEY INDEX index_name
      [ ON <catalog_filegroup_option> ]
      [ WITH [ ( ] <with_option> [ ,...n] [ ) ] ]
[;]
```

[4] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

Cria um índice de *FULL-TEXT* na tabela ou *view* indexada *table_name* sobre a coluna *column_name*. Se na lista de colunas indicar-mos colunas do tipo *varbinary*, *varbinary(max)* ou *image* para guardar ficheiros então podemos indicar o tipo de ficheiro noutra coluna, para isso usamos a clausula *TYPE COLUMN type_column_name*. A coluna *type_column_name* tem de ser do tipo *char*, *nchar*, *varchar* ou *nvarchar*.

A clausula *LANGUAGE* serve para indicar qual a coluna esta guardada o idioma do tuplo.

A clausula *KEY INDEX* serve para indicar um índice único para ser chave do índice *FULL-TEXT*.

A clausula *ON* serve para indicar-mos em que *filegroup* e/ou que *catalog* é armazenado o índice.

A clausula *WITH* serve para indicar-mos opções da criação do índice. As opções são as seguintes:

- **CHANGE_TRACKING [=] { MANUAL | AUTO | OFF [, NO POPULATION] }**: Especifica quando são feitas alterações feitas na tabela e esta forem abrangidas pelo o índice, estas serão propagadas do SQL Server até ao índice de *FULL-TEXT*
- **STOPLIST [=] { OFF | SYSTEM | stoplist_name }**: Associa uma *STOPLIST* ao índice. Uma *STOPLIST* é uma lista de palavras comuns, para impedir que o índice fique cheio destas palavras.

4.1.5 Remoção de Índices[17]

A sintaxe[5] para remover um índice relacional, *SPATIAL* ou XML é:

```
DROP INDEX
{ <drop_relational_or_xml_or_spatial_index> [ ,...n ]
}
```

Este comando remove um índice relacional, *SPATIAL* ou XML com nome *<drop_relational_or_xml_or_spatial_index>*.

4.1.6 Remoção de Índices *FULLTEXT*[16]

A sintaxe[6] para remover um índice *FULLTEXT* é:

```
DROP FULLTEXT INDEX ON table_name
```

Este comando remove um *FULLTEXT* da tabela *table_name*.

[5] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

[6] Para mais detalhe sobre as conversões da sintaxe consulte `http://msdn.microsoft.com/en-us/library/ms177563.aspx`

4.2 Estruturas de Dados

O SQL Server para implementar índices só implementa árvores B+. Em termos de organização, as árvores em SQL Server estão organizadas em:

- *CLUSTERED*
- *NONCLUSTERED*

4.2.1 *CLUSTERED Index*[8]

Nos índices *CLUSTERED*, as folhas da árvore contêm os dados da tabela que estamos a indexar. A raiz da árvore e os níveis intermédios contêm páginas de valores indexados. Cada valor indexado contem um apontador para outro nível intermédio ou para as folhas. As páginas de cada nível estão ligadas por uma *double linked list*.

Por defeito, os índices *CLUSTERED* têm só uma partição. Quando um índice *CLUSTERED* tem várias partições, cada partição contem uma árvore B+ com os dados dessa partição.

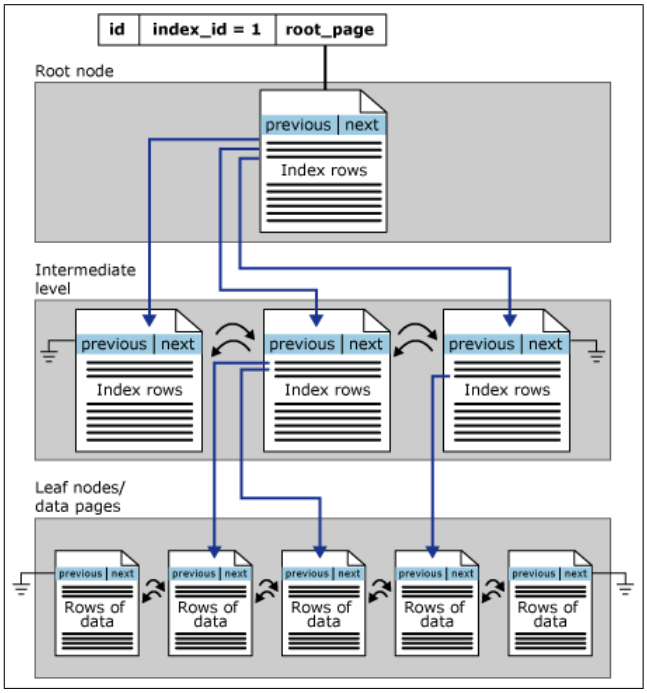


Figura 4.1: *Clustered Index*

4.2.2 *NONCLUSTERED Index*[24]

Os índices *NONCLUSTERED* possuem a mesma estrutura que os índices *CLUSTERED*, excepto nas seguintes diferenças:

- Os dados da tabela indexada não são guardados ou ordenados pela a ordem da sua chave do índice
- As folhas de um índice *NONCLUSTERED* são referencias para as páginas de dados

Um índice *NONCLUSTERED* pode ser definido numa tabela ou *view* com um índice *CLUSTERED* ou com *Heap*. Cada valor num índice *NONCLUSTERED* contem a chave e um localizador do valor. O localizador aponta para dos dados que estão um índice *CLUSTERED* ou numa *Heap*.

Para criar numa tabela este tipo de índices, esta tem de ter previamente um índice *CLUSTERED*.

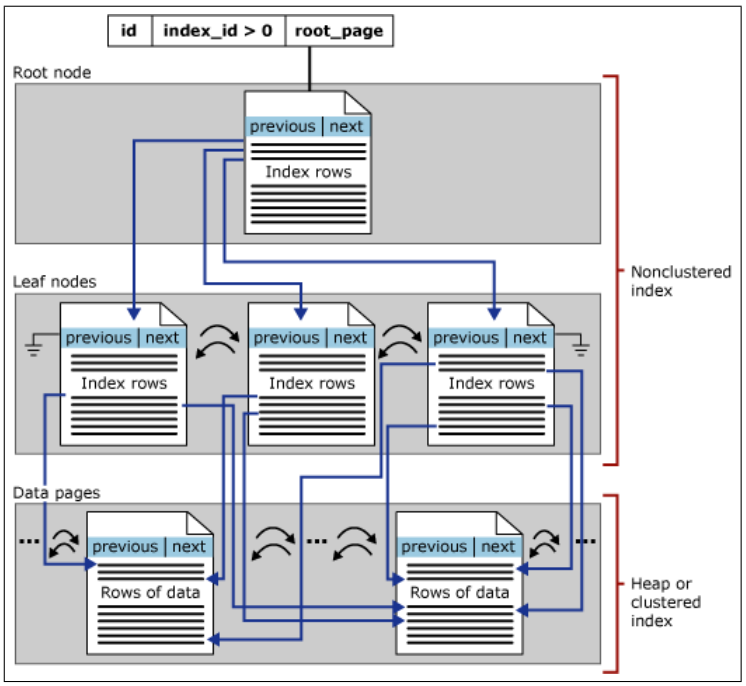


Figura 4.2: *NONCLUSTERED Index*

Capítulo 5

Processamento e optimização de perguntas[6]

O SQL Server utiliza várias técnicas para minimizar o custo das perguntas (queries), e estas técnicas determinam a estratégia de processamento das perguntas. O optimizador considera vários factores para decidir sobre quais os indexes e operações de junção utilizará no processamento da pergunta, factores esses que são baseados em estatísticas e meta dados. Minimizando o impacto desses factores no custo final da execução da pergunta, é criado um plano, que é o plano que minimiza o custo de todas as operações envolvidas no processamento da pergunta.

A seguinte imagem indica as técnicas que o SQL Server utiliza para optimizar as perguntas e criar o respectivo plano de execução para a pergunta.

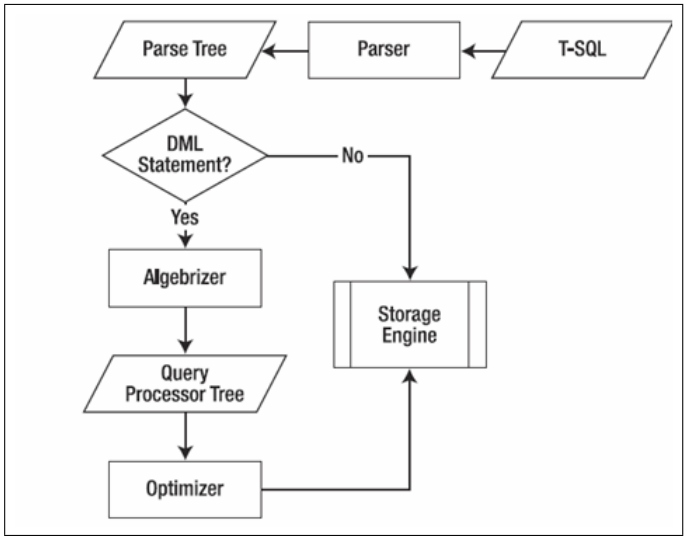


Figura 5.1: *Passos para criação dos planos de execução*

- Parser: O parser valida a sintaxe da pergunta. Se a sintaxe estiver errada, a pergunta ou uma sequencia de perguntas não são executadas. Se a sintaxe estiver correcta, o parser cria uma estrutura chamada parse tree para ser consumida pelo componente Algebrizer.

- Algebrizer: O Algebrizer verifica se as tabelas e respectivos atributos referenciados na pergunta existem na base de dados. O Algebrizer retorna uma estrutura chamada Query Processor Tree, que é uma representação lógica dos passos necessários para a execução do comando SQL. O Query Processor Tree é enviado para o próximo passo da execução da consulta, que é a análise do Query Optimizer. Nem sempre o resultado obtido pelo Algebrizer é enviado para o Query Optimizer. Um exemplo disso é a criação de uma tabela. Não faz sentido enviar o resultado desse comando para o Optimizer, pois só existe uma única maneira de executar o comando SQL.

- Optimizer: O query optimizer é o componente principal para escolher o melhor plano para a execução de uma pergunta. Ele escolhe que algoritmos de junção, selecção, etc. que podem ser utilizados para minimizar o custo da execução, e se a pergunta pode ser dividida e ser executada em paralelo (isto se houver mais que 1 processador). Para escolher o melhor plano, o Optimizer recorre a dados estatísticos.

5.0.3 Query Optimizer

Existem várias maneiras de executar a perguntar presente na *Query Processor Tree*. Essas maneiras são calculadas e o custo de cada uma delas é comparado com todas as outras para achar o plano de execução de menor custo. O Query Optimizer (ou só Optimizer) adopta várias técnicas para encontrar o melhor plano de execução. As técnicas usadas são:

- Igualar planos de execução - O SQL Server mantém em sua posse uma lista de planos triviais de execuções de perguntas. Ele verifica se já tem na lista um plano de execução para essa pergunta, e se tiver, é criado um plano para a pergunta sem qualquer optimização.

- Multiplas Fases de Optimização - Usando esta técnica, o optimizer tenta encontrar as melhores técnicas de junção e de índices para essa pergunta. A técnica usa estatisticas associadas às colunas referidas nas cláusulas WHERE para avaliar a melhor maneira de como fazer as pesquisas (usando índices ou não) e quais as melhores estratégias de junção.

- Paralelismo - Falado mais à frente neste relatório.

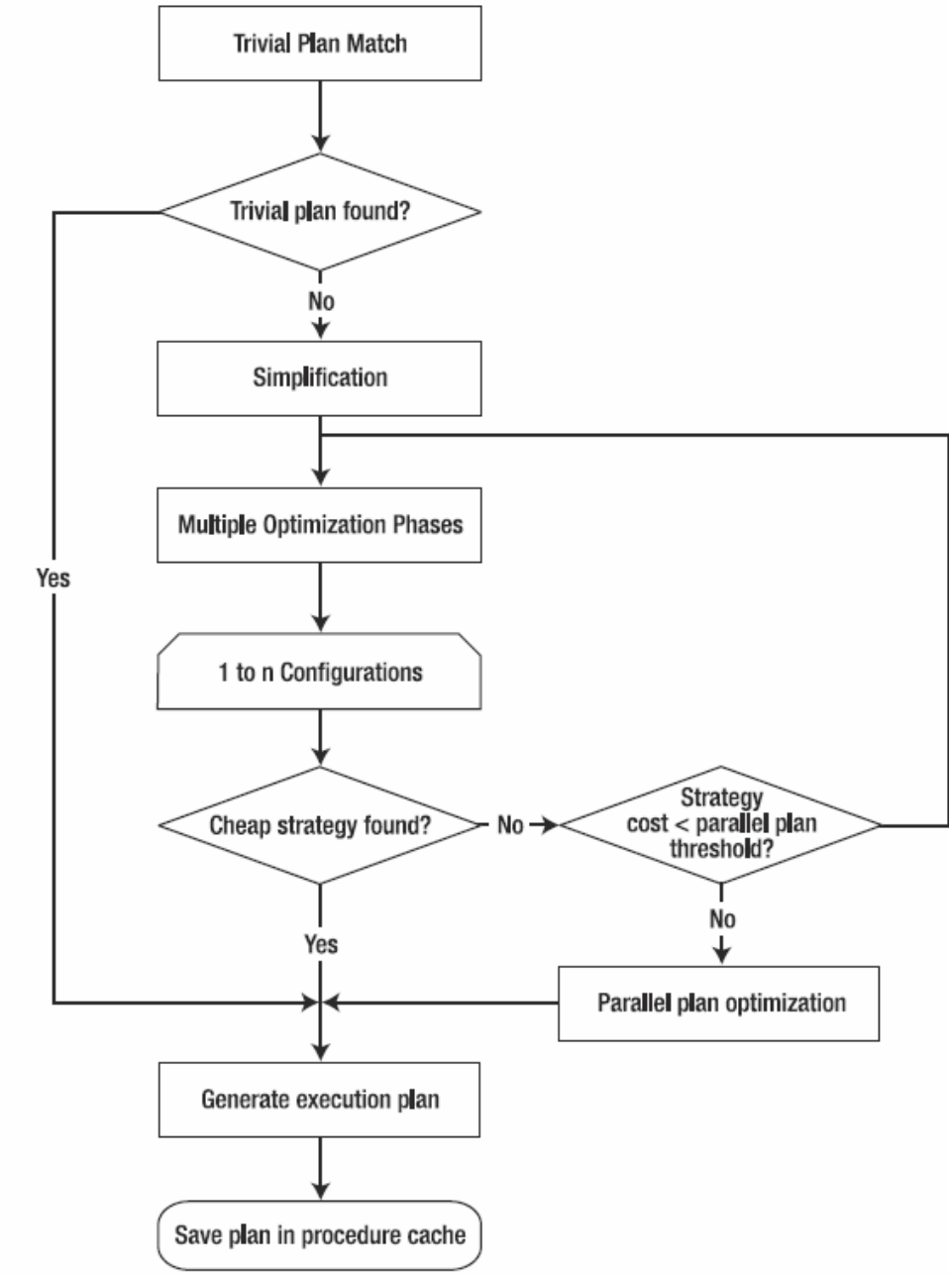


Figura 5.2: *Passos do Optimizer*

5.1 Algoritmos implementados

5.1.1 Junção

O SQL Server implementa 3 tipos de algoritmos de junção:

- Hash join;
- Merge join;
- Nested Loop join.

Hash Join[1]

Em baixo descrevemos brevemente o algoritmo, apresentando o pseudo-código.

```
for all row R1 in the build table do
  calculate hash value on R1 join key(s)
  insert R1 into the appropriate hash bucket
end for
for all row R2 in the probe table do
  calculate hash value on R2 join key(s)
  for all row R1 in the corresponding hash bucket do
    if R1 joins with R2 then
      return (R1,R2)
    end if
  end for
end for
```

A build table é a tabela com menos tuplos.

É criada uma hash table em memória. O algoritmo percorre tuplo a tuplo da build table e calcula uma valor de hash através da chave de junção das tabelas, e insere o respectivo tuplo no bucket corresponde ao valor de hash. O algoritmo depois calcula para cada tuplo da probe table o seu valor de hash, e verifica se existe um bucket associado e encontra o tuplo que combina. Se for encontrado, retorna o resultado dessa junção para o output.

A imagem abaixo mostra os passos usamos no algoritmo.

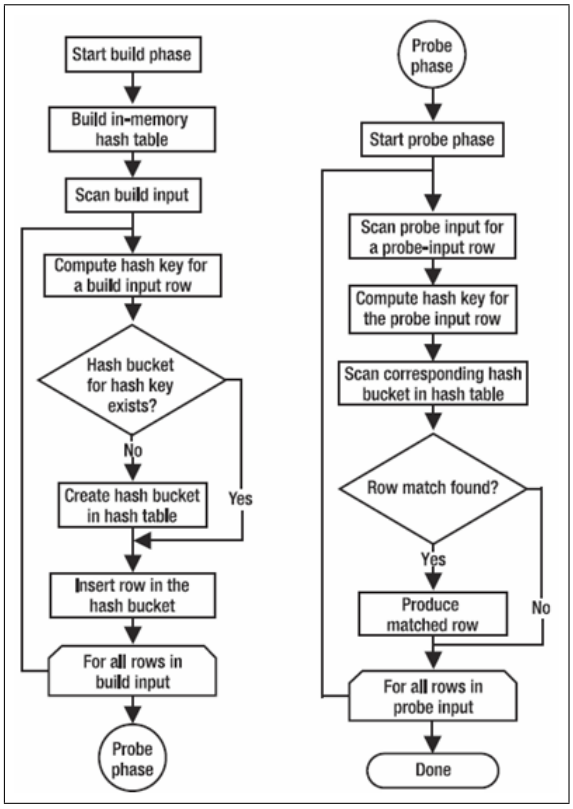

Start build phase
Build in-memory hash table
Scan build input
Compute hash key for a build input row
Hash bucket for hash key exists?
No
Yes
Create hash bucket in hash table
Insert row in the hash bucket
For all rows in build input
Probe phase
Probe phase
Start probe phase
Scan probe input for a probe-input row
Compute hash key for the probe input row
Scan corresponding hash bucket in hash table
Row match found?
Yes
No
Produce matched row
For all rows in probe input
Done

Figura 5.3: *Hash Join*

Merge join[2]

O SQL Server implementa o Merge join de igual forma ao Oracle.

O algoritmo merge join requer que ambas as tabelas estejam ordenadas pelo atributo de junção. O algoritmo utiliza as duas tabelas em simultâneo, na medida em que obtém o primeiro tuplo de cada tabela de input, e compara se o atributo de junção de cada tuplo é igual. Se for, então retorna a junção dos 2 tuplos, e obtém o 2º tuplo da 2ª tabela de input, e repete os passos acima descritos. Se não forem iguais, e se o 1º tuplo da 1ª tabela de input for menor que o 1º tuplo da 2ª tabela de input, então obtém o próximo tuplo da ª tabela de input e repete os passos atrás. Se for maior, então o obtém o próximo tuplo da 2ª tabela de input.

Para melhor compreensão, em baixo segue o pseudo-código do algoritmo.

```
get first row R1 from input 1
get first row R2 from input 2
while not at the end of either input do
  if R1 joins with R2 then
    return (R1,R2)
    get next row R2 frim input 2
  else
    if R1 < R2 then
      get next row R1 frim input 1
    else
      get next row R2 frim input 2
    end if
  end if
end while
```

De realçar que as tabelas podem estar ordenadas pela ordem natural do atributo de junção, ou então poderá usar-se um índex sobre esse atributo. Em geral, é usado o índex desse atributo, pois tem um custo menor.

Nested Loop-Join[3]

A implementação deste algoritmo é exactamente igual ao implementado pelo Oracle.

O pseudo-código segue abaixo.

```
for all row R1 in the outer table do
  for all row R2 in the inner table do
    if R1 joins with R2 then
      return (R1,R2)
    end if
  end for
end for
```

Uma forma de melhorar o algoritmo é criar um índex sobre o atributo da junção numa das tabelas. Imaginando o seguinte exemplo:

```
create table Clientes (client_id int, nome varchar(10))
insert Customers values (1, 'Diogo')
insert Customers values (2, 'Pedro')
insert Customers values (3, 'Joao')

create table Vendas (client_id int, produto varchar(10))
insert Sales values (2, 'Camera')
insert Sales values (3, 'Computador')
insert Sales values (3, 'Monitor')
insert Sales values (4, 'Impressora')
```

Se previamente existir um índex na tabela 'Vendas' sobre o atributo client_id, o plano de execução escolherá usar uma pesquisa por índex em vez de percorrer a tabela inteira, ou seja, percorre-se cada tuplo da tabela Clientes, e para cada um deles, procurar pela correspondência na tabela Vendas, usando um índex seek.

5.2 Mecanismos para expressões complexas

5.2.1 Paralelismo[4]

O SQL Server permite execução de perguntas em paralelo, quando possível e necessário, para optimização de perguntas. Durante o processo de optimização, o query optimizer procura pelas perguntas que possam ser executadas em paralelo para criar um plano adequado. Claro que só irá procurar se houver mais que 1 CPU no Servidor, se houver memória suficiente, entre mais outros factores, caso contrário o optimizer só considerará um único CPU.

Para permitirmos que mais que 1 CPU seja usado para o processamento de perguntas, devemos alterar uma configuração do SQL Server, que é chamada de affinty mask.A affinty mask é na realidade um bitmap, onde o número de 1's é o número de CPU's que queremos que fiquem disponíveis para a execução das perguntas. Podemos alterar o número de CPU's, executando os seguintes comandos:

```
USE master
EXEC sp_configure 'show advanced option', '1'
RECONFIGURE
EXEC sp_configure 'affinity mask' , 15 -bit map: 0001111
RECONFIGURE
```

Após a execução destes comandos, estou a indicar que quero usar 4 CPU's para a execução das perguntas.

Podemos definir o número de CPU's usados apenas para o paralelismo. Isto é, se tiver 4 CPU's e representados na affinity mask, podemos apenas definir 2 CPU's para o paralelismo de perguntas. Isso pode ser feito, executando os seguintes comandos:

```
USE master
EXEC sp_configure 'show advanced option', '1'
RECONFIGURE
EXEC sp_configure 'max degree of parallelism' , 2
RECONFIGURE
```

5.2.2 Materialização

O SQL Server implementa materialização. Quando uma operação requer que os dados de entrada sejam todos usados antes de retornar o resultado para o operador pai, é necessário guardar esses dados para serem consumidos pela operação.

Infelizmente não encontrámos muito mais informação acerca deste tema, mas conseguimos saber que o SQL Server faz uso deste mecanismo.

5.2.3 Pipelining

O SQL Server implementa pipelining e usa pipelining para certos operadores, como projecções, selecções e junções. Isto permite que os resultados obtidos sejam enviados para o output muito mais rapidamente.

Infelizmente não encontrámos muito mais informação acerca deste tema, mas conseguimos saber que o SQL Server faz uso deste mecanismo.

5.3 Estatísticas[6]

O SQL Server mantém estatísticas sobre índices de colunas actualizadas. Esta opção está ligada por omissão. Para se poder desligar esta opção, temos que fazer:

```
ALTER DATABASE universidade
SET AUTO_UPDATE_STATISTICS OFF;
```

Para optimizar essas actualizações o SQL Server usa um algoritmo para actualizar as estatísticas que é executado quando existe um grande número de modificações na determinada coluna ou quando o tamanho da tabela aumenta. Por exemplo, o SQL Server actualiza as estatísticas quando uma tabela que não tinha tuplos, passa a ter mais que 1; quando o tamanho da tabela é menor que 500 tuplos e aumenta, pelo menos, para o dobro; e quando uma tabela que tenha mais que 500 tuplos aumenta em 20% da sua capacidade. Nestas situações, as estatísticas são actualizadas.

As estatísticas também podem ser actualizadas assincronamente. Isto é, as consultas são compiladas com estatísticas existentes mesmo que as estatísticas estejam desactualizadas. O query optimizer poderá escolher um plano de execução com maior custo se as estatísticas estiverem desactualizadas na compilação da consulta. Se essa consulta provoca actualização das estatísticas, essas estatísticas poderão ser actualizadas depois de a consulta ser executada. Se fosse feito de forma síncrona, isto não aconteceria, ou seja, o optimizador procuraria sempre actualizar as estatísticas antes da execução da pergunta. A opção de actualizar as estatísticas de forma assíncrona está desactivada por omissão. O SQL Server também cria estatísticas para atributos sem índices associados. Essa opção está activa por omissão, mas poderá ser desactivada. É aconselhável ter essa opção activa. Só faria sentido desactivar, se não valesse a pena, tal como fazer sempre consultas diferentes (não habitual).

5.3.1 Tipo de Estimativas

O SQL Server guarda várias informações relativamente às estatísticas de uma tabela. Ele guarda informação sobre quantas linhas tem a tabela, a última data de actualização das estatísticas, o tamanho médio da chave, a densidade de combinações das colunas.

O SQL Server guarda também várias estruturas internas chamadas histogramas. Esta estrutura consiste na amostra de uma distribuição de dados sobre uma coluna ou indexes. Para se perceber um pouco melhor, o histograma é um gráfico de barras, onde no intervalo X, o intervalo entre X e

X+1 chamado de passo, que pode ser interpretado como um intervalo de valores. O intervalo Y representa a amostra para esse intervalo X. Imaginemos uma tabela com 2 atributos inteiros, a chave e um valor, podemos imaginar que seja o saldo de uma conta e a chave o número da conta. Imaginemos que é criado um histograma para estatísticas sobre o saldo. O intervalo entre X e X+1 representa um conjunto de contas, e o valor Y é uma amostra. Imaginemos que no histograma temos um passo de 10000, isto é, o intervalo de X (número da conta) do histograma será [0; 10000; 20000; 30000; 40000], e para cada passo teremos informação acerca do maior saldo nas contas nesse intervalo de número de contas, o valor médio do saldo, o número de contas com o maior saldo e o número de saldos diferentes. O SQL Server utiliza estas estatísticas para poder criar um plano de menor custo.

5.3.2 Algumas considerações

- AUTO_CREATE_STATISTICS: Quando esta opção está ligada, o optimizador de perguntas cria estatísticas para os atributos de cada predicado, se for necessário. Por omissão esta opção está ligada.
- AUTO_UPDATE_STATISTICS: Quando esta opção está ligada, o optimizador de perguntas define quando as estatísticas podem estar desactualizadas, e quando deve actualizar as estatísticas. Ele actualiza as estatísticas com base na quantidade de novos tuplos ou na quantidade de dados alterados. Por omissão esta opção está ligada.
- AUTO_UPDATE_STATISTICS_ASYNC: Quando esta opção está ligada, o optimizador de perguntas actualiza as estatísticas de forma assíncrona. Por omissão esta opção está desligada.
- EXEC sp_updatestats: Executando este comando, o administrador da base de dados actualiza as estatísticas. Esta costuma ser uma boa opção, mas temos que ter em conta que isto obriga a que consultas compiladas, terão que ser recompiladas com base na actualização. Convém aplicar esse comando quando as perguntas de consulta começam a ficar muito lentas.
- EXEC sp_autostats 'alunos': Mostra a ultima actualização de estatísticas acerca dos atributos da tabela 'alunos'.

5.3.3 Exemplos

No exemplo abaixo mostramos como um administrador pode actualizar as estatísticas. Para isso usa o comando sp_updatestats.

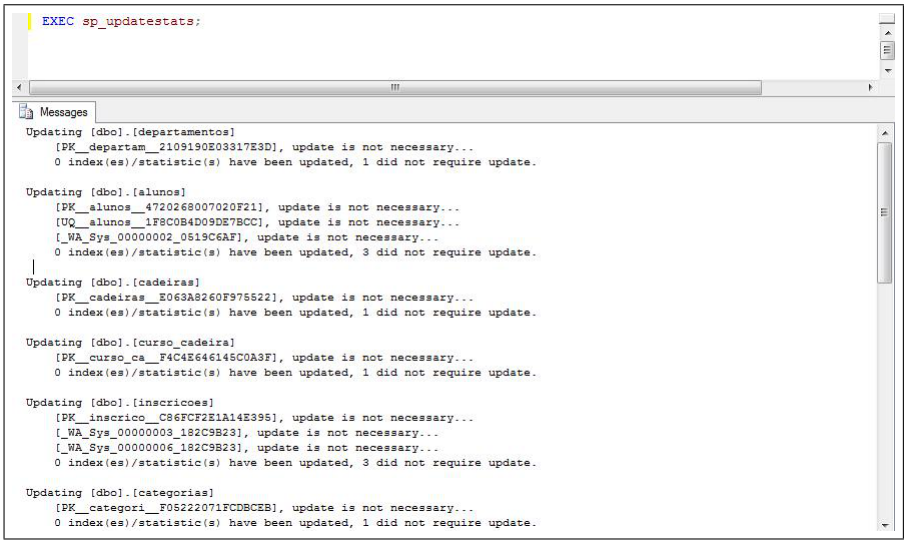

EXEC sp_updatestats;
Messages
Updating [dbo].[departamentos]
[PK__departam__2109190E03317E3D], update is not necessary...
0 index(es)/statistic(s) have been updated, 1 did not require update.
Updating [dbo].[alunos]
[PK__alunos__4720268007020F21], update is not necessary...
[UQ__alunos__1F8C0B4D09DE7BCC], update is not necessary...
[_WA_Sys_00000002_0519C6AF], update is not necessary...
0 index(es)/statistic(s) have been updated, 3 did not require update.
Updating [dbo].[cadeiras]
[PK__cadeiras__E063A8260F975522], update is not necessary...
0 index(es)/statistic(s) have been updated, 1 did not require update.
Updating [dbo].[curso_cadeira]
[PK__curso_ca__F4C4E646145C0A3F], update is not necessary...
0 index(es)/statistic(s) have been updated, 1 did not require update.
Updating [dbo].[inscricoes]
[PK__inscrico__C86FCF2E1A14E395], update is not necessary...
[_WA_Sys_00000003_182C9B23], update is not necessary...
[_WA_Sys_00000006_182C9B23], update is not necessary...
0 index(es)/statistic(s) have been updated, 3 did not require update.
Updating [dbo].[categorias]
[PK__categori__F05222071FCDBCEB], update is not necessary...
0 index(es)/statistic(s) have been updated, 1 did not require update.

Figura 5.4: *Comando updatestats*
Como se pode verificar pelo exemplo, as actualizações não foram necessárias.

Executando o comando sp_autostats, verificamos quando foi a última actualização.

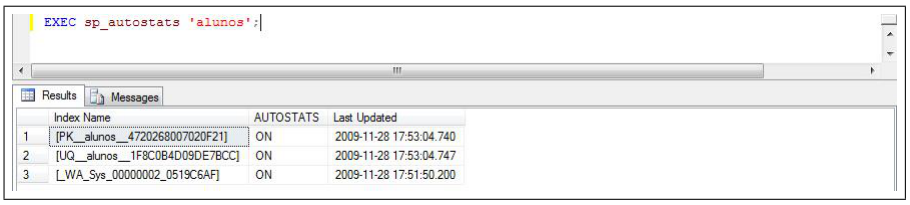


Figura 5.5: *Comando autostats*

Abaixo segue um exemplo de como um administrador pode verificar se as opções de criar estatísticas e actualizar estão ligadas. Se o resultado da consulta for 1, então é porque estão ligadas.

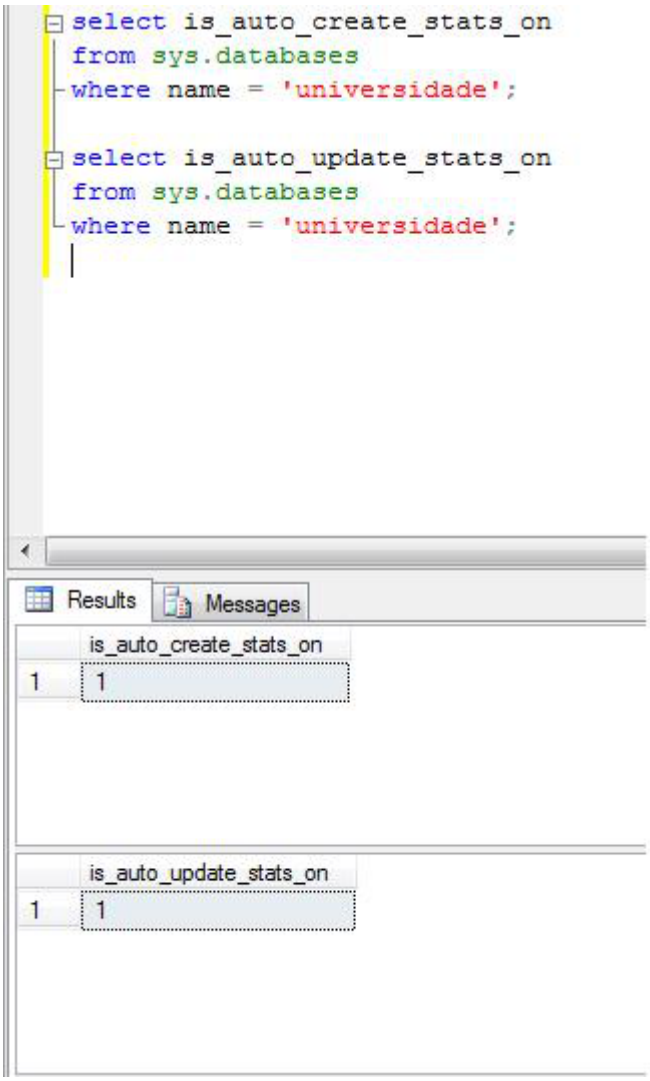


Figura 5.6: *Verificar se a criação de estatísticas esta ligada.*

5.4 Caching de planos de execução[5]

Antes da execução de uma pergunta, o SQL Server cria um plano de execução que minimiza os custos da sua execução. O SQL Server fornece um mecanismo para guardar planos de execução em cache para serem reutilizados mais tarde, se possível.

Os planos não são guardados para sempre, pois não é boa política guardar planos de execução que sejam usados e reutilizados durante um pequeno período de tempo, e depois nunca mais sejam utilizados. O SQL Server para descartar estes planos de execução recorre a um método muito simples. Ele atribui um valor a esse plano de execução denominado de age. Este valor é atribuído no momento da sua criação e o seu valor numérico depende da complexidade da pergunta, ou seja, uma pergunta simples tem um valor age muito mais pequeno que numa pergunta complexa. Se uma pergunta reutilizar um dos planos em cache, o valor actual age é incrementado pelo valor age que é lhe atribuído. Para entender melhor vamos descrever um pequeno exemplo. Imagine uma query1 e outra query2 e os respectivos planos de execução guardados em cache. Vamos supor que a query1 é menos complexa que a query2, logo o valor age da query1 é menor que o valor age da query2. Vamos supor que o valor age de query1 é 10, e o valor age da query2 é 100. Se uma nova pergunta à Base de Dados for feita e reutilizar o plano de execução da pergunta 1, o valor age irá ser incrementado para 20 (10 valor actual + 10 valor atribuído). O mesmo se passará para a query2. O valor age irá sendo decrementado ao longo do tempo e quando chegar a 0, o plano de execução passa a ser um candidato a ser removido da cache. O SQL Server só remove os planos com valor age 0 se necessitar de os remover. É importante de referir que planos de execução com custos extremamente reduzidos não serão guardados em cache.

Há duas categorias de perguntas no armazenamento de planos de execução:

- Adhoc Workload
- Prepared Workload

Na categoria Adhoc, as perguntas são guardadas sem isolamento explícito das variáveis, isto é, por exemplo as variáveis da cláusula where são guardadas no plano em vez de serem parametrizadas. Imaginemos o seguinte exemplo:

```
SELECT *
FROM alunos
WHERE nome = 'diogo';
```

O plano de execução para esta pergunta é guardado em cache com a variável implícita, isto implica que se fizer a mesma pergunta mas mudando

a variável para 'joao', o SQL Server iria guardar esse plano de execução em vez de reutilizar o que estava em cache.

Há maneira de parametrizar as variáveis e existem 3 técnicas para o fazer que se inserem na categoria Prepared Workload. Nesta categoria, as variáveis das perguntas são parametrizadas, o que permite que possamos usar o mesmo plano de execução para uma pergunta, se mudarmos apenas as variáveis que estão parametrizadas. Existem 3 métodos para submeter as perguntas desta forma:

- Armazenamento de procedimentos.
- Executando comando T-SQL sp_executesql.
- Preparar/executar modelo.

Armazenamento de procedimentos - Quando um procedimento é criado e compilado é criado um plano de execução, em que as variáveis são parametrizadas.

Comando T-SQL sp_executesql - É um mecanismo que permite submeter uma ou mais perguntas com as variáveis parametrizadas fornecendo a reutilização dos planos de execução como procedimentos.

Preparar/executar modelo - Não iremos descrever nem mostrar exemplos, pois não achamos relevante.

5.4.1 Exemplo Adhoc Workload

Para exemplificar o armazenamento de planos de execução Adhoc, fizemos a seguinte pergunta, por 4 vezes:

```
SELECT  *
FROM alunos
WHERE cod_curso = 1;
```

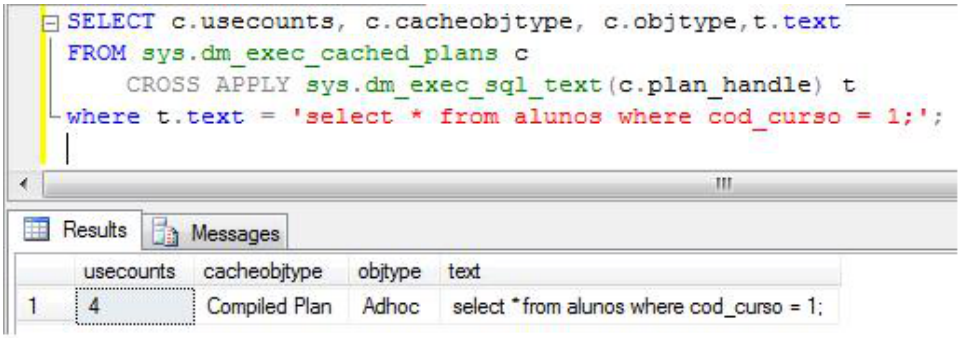


Figura 5.7: *Adhoc*

Como se pode verificar, esta pergunta (com exactamente as mesmas variáveis) foi feita por 4 vezes. Se fizermos esta mesma pergunta, mas mudando a variável, então será criado um novo plano de execução e guardado em cache.

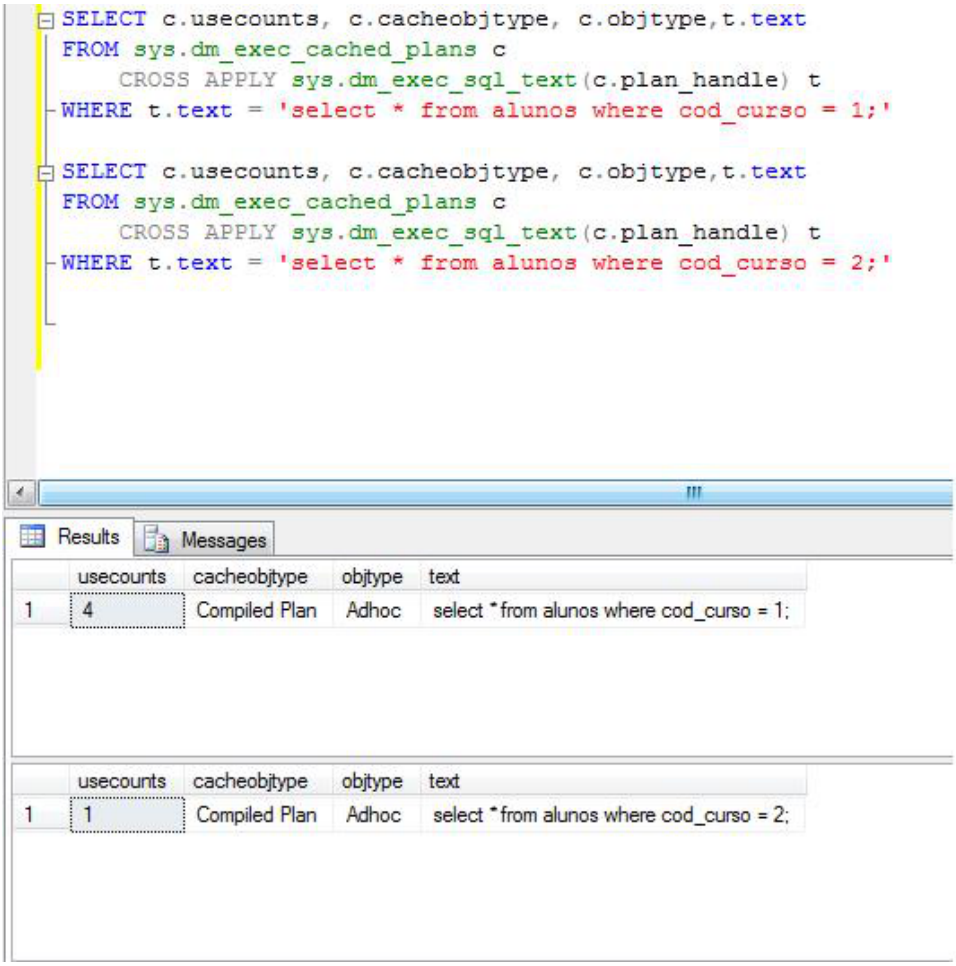

SELECT c.usecounts, c.cacheobjtype, c.objtype,t.text
FROM sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE t.text = 'select * from alunos where cod_curso = 1;'
SELECT c.usecounts, c.cacheobjtype, c.objtype,t.text
FROM sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE t.text = 'select * from alunos where cod_curso = 2;'
Results
Messages
usecounts cacheobjtype objtype text
1 4 Compiled Plan Adhoc select * from alunos where cod_curso = 1;
usecounts cacheobjtype objtype text
1 1 Compiled Plan Adhoc select * from alunos where cod_curso = 2;

Figura 5.8: *Adhoc*

5.4.2 Exemplo Prepared Workload

Neste exemplo fazemos a mesma pergunta do exemplo Adhoc. O plano será criado com as variáveis parametrizadas, isto significa que quando fizermos uma mesma pergunta com outras variáveis, o SQL Server irá utilizar este plano de execução. Portanto, fizemos as seguintes 3 perguntas:

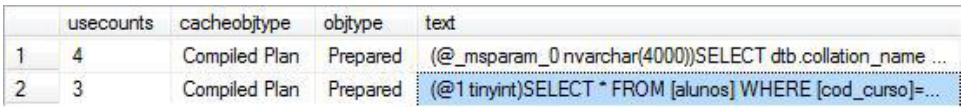
```
SELECT *
FROM alunos
WHERE cod_curso = 1;

SELECT *
FROM alunos
WHERE cod_curso = 2;

SELECT *
FROM alunos
```

```
WHERE cod_curso = 3;
```

E o resultado é este (olhando apenas para a 2ª linha): Como se pode verificar



	usecounts	cacheobjtype	objtype	text
1	4	Compiled Plan	Prepared	(@_msparam_0 nvarchar(4000))SELECT dtb.collation_name ...
2	3	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [alunos] WHERE [cod_curso]=...

Figura 5.9: *Prepared*

o plano de execução foi utilizado por 3 vezes. (O numero de vezes que foi executada a mesma pergunta, apesar de utilizar variáveis diferentes).

5.5 Mecanismos para ver planos de execução

Usando o Microsoft Management Studio podemos ver os planos de execução criados para uma pergunta. Temos a oportunidade de os ver graficamente, como texto e em formato XML.

5.5.1 Graficamente

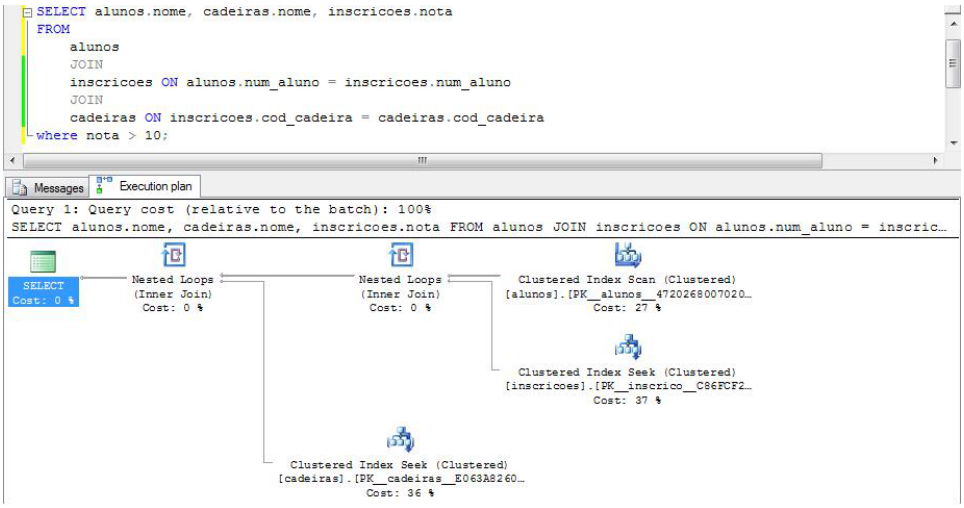


Figura 5.10: *Visualização gráfica*

5.5.2 Formato XML

Figura 5.11: *XML*

5.5.3 Texto

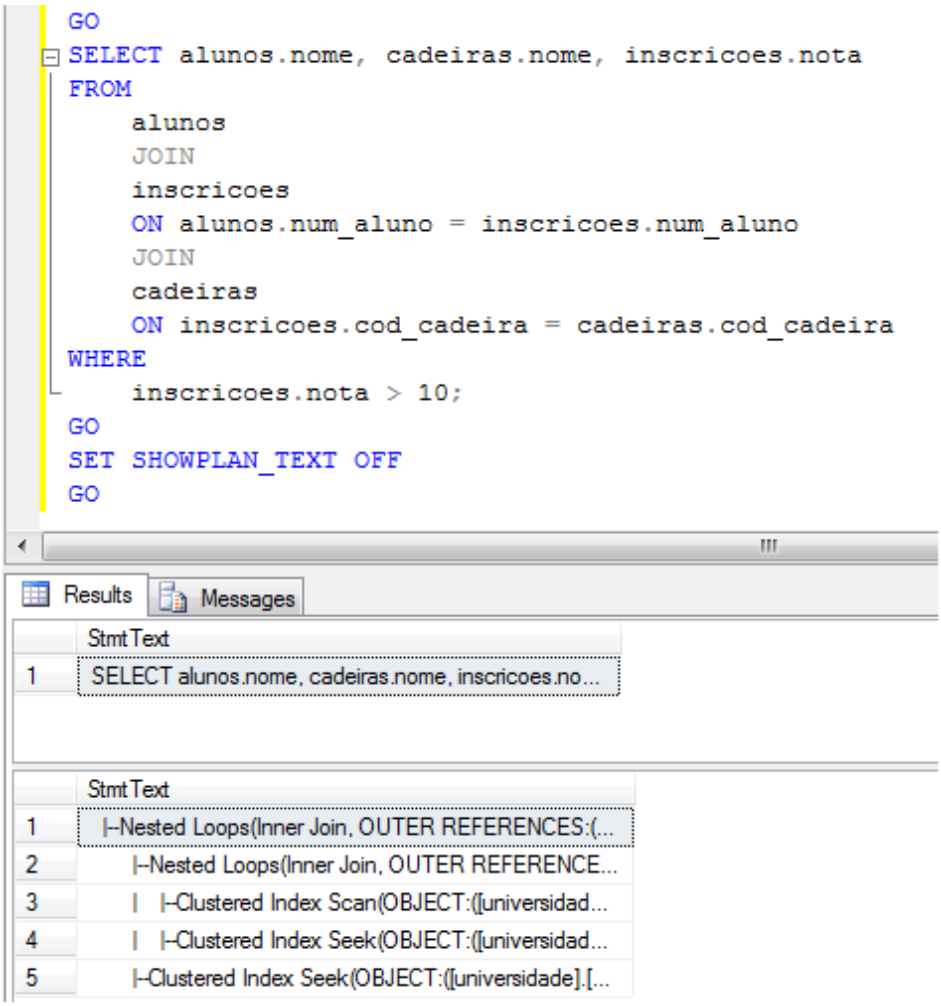


Figura 5.12: *Texto*

Capítulo 6

Gestão de transacções e controlo de concorrência

Uma transacção é uma sequência de operações sobre dados existentes numa base de dados, em que essa sequência de operações é do ponto de vista lógico uma única operação. Para isto ser garantido a transacção tem que obedecer a 4 propriedades. Atomicidade, Consistência, Isolamento e Durabilidade. Frequentemente designadas como propriedades A.C.I.D.

- Atomicidade - Para uma transacção ser considerada atómica, todas as operações realizadas pela transferência são executadas com sucesso ou nenhuma delas é executada.
- Consistência - Quando a transacção é efectuada, os dados têm que continuar consistentes. Isto é, a transacção não pode modificar a estrutura interna do modelo de dados de forma a que os dados fiquem inconsistentes.
- Isolamento - Esta propriedade garante que uma transacção ao ser executada concorrentemente com outra, ela execute com dados consistentes, isto é, a transacção só irá operar sobre os dados anteriores à realização da outra transacção ou apenas com os dados actualizados após a outra transacção, e nunca opere sobre dados num estado intermédio.
- Durabilidade - Esta propriedade garante que após a realização da transacção os efeitos da transacção permaneçam no sistema, mesmo que o sistema falhe.

6.1 Modos de transacção

O SQL Server suporta vários modos de transacção. Os modos que suporta são os seguintes:

- Transacções AutoCommitted
- Transacções Implícitas
- Transacções Explicitas

6.1.1 Transacções AutoCommitted[34]

Este é o modo por omissão do SQL Server. Neste modo uma instrução T-SQL que não falhe é automaticamente *committed* após sucesso,ou *rolledback* caso falhe. Este modo é usado para todas as transacções, menos quando é imposto o modo implícito ou explicito sobre as transacções. O modo AutoCommitted volta a operar até um dos outros dois modos for desligado. No caso do modo explicito, quando a transacção operar nesse modo, o modo de transacção volta a ser AutoCommitted quando a transacção em modo explicito fizer *commit transaction* ou *rollback transaction*.

6.1.2 Transacções Implícitas[36]

No modo implicito de transacções não se define o inicio da transacção, apenas o fim dela aplicando *COMMIT*. Neste modo, cada operação inicia uma nova transacção. A execução desta é efectuada após a transacção anterior terminar. O conjunto de instruções DDL e DML que iniciam a transacção neste modo estão descritas na tabela abaixo.

ALTER TABLE	INSERT	CREATE	OPEN
DELETE	REVOKE	DROP	SELECT
FETCH	TRUNCATE TABLE	GRANT	UPDATE

Exemplo de transacção em modo implícito

```
SET IMPLICIT_TRANSACTIONS ON
GO

declare @aluno int;
declare @cadeira int;

/* Obter o numero do aluno em questao. */
SET @aluno =
(select alunos.num_aluno
from alunos
where alunos.nome LIKE 'Paula Antunes')

/* Obter o codigo da cadeira para actualizacao da nota. */
SET @cadeira = (
select cadeiras.cod_cadeira from cadeiras where cadeiras.nome
    LIKE 'Programacao 1');

/* Actualizar a nota. */
UPDATE inscricoes
SET nota = 10
where inscricoes.num_aluno = @aluno and inscricoes.cod_cadeira =
     @cadeira;
GO

COMMIT TRANSACTION;

/* Voltar ao modo Auto-Committed */
SET IMPLICIT_TRANSACTIONS OFF
GO
```

6.1.3 Transacções Explicitas[35]

Neste modo de transacção, quem cria a transacção define explicitamente quando a transacção começa (escrevendo *BEGIN TRANSACTION*) e indica também explicitamente quando acaba (escrevendo *COMMIT TRANSACTION*). É um modo útil de executar uma transacção, pois se uma transacção for efectuada várias vezes ao longo do tempo, o utilizador pode iniciar essa transacção que já está criada, e assim não precisa de escrever novamente todas as operações.

Exemplo de criação de transacção explicita

```
BEGIN TRANSACTION actualizar_nota
declare @aluno int;
declare @cadeira int;

SET @aluno =
(select alunos.num_aluno
from alunos
where alunos.nome LIKE 'Paula Antunes')

SET @cadeira = (
select cadeiras.cod_cadeira from cadeiras where cadeiras.nome
    LIKE 'Programacao 1');

UPDATE inscricoes
SET nota = 18
where inscricoes.num_aluno = @aluno and inscricoes.cod_cadeira =
     @cadeira;

COMMIT TRANSACTION actualizar_nota;
```

6.2 Tópicos avançados sobre Transacções

6.2.1 Transacções Nested[37]

O SQL Server suporta transacções nested em transacções explicitas, isto significa, que suporta criação de transacções dentro de outras transacções. Assim sendo, existirão vários níveis de transacções. A transacção 'pai' é designada como *outer transaction* e a transacção 'filho' é designada por *inner transaction*. A necessidade deste tipo de transacções é importante para se poder definir procedimentos que contêm transacções e posteriormente poder executa-los numa outra transacção tornando assim o código mais extensível.

6.2.2 Transacção de longa duração[38]

O SQL Server 2008 tem suporte para transacções de longa duração. Ele utiliza mecanismo de logging virtual para manter as actividades activas. Este logging virtual contém o primeiro log de registo da transacção.

Por vezes poderá ser necessário parar uma transacção e podemos fazer isso fazendo kill [numero do processo].

Um dos grandes problemas deste mecanismo é o facto de o log ficar demasiado grande e é preciso tratamento caso isso aconteça.

6.2.3 Savepoints[29]

O SQL Server tem um mecanismo de *savepoints* em que permite guardar porções de transacções e com isso poderá voltar atrás na transacção até ao sitio onde foi feito o savepoint, ou seja, se fizermos rollback, podemos voltar até ao ponto gravado, em vez de voltarmos ao inicio da transacção. É importante referir que quando é aplicado um savepoint, os recursos modificados antes do savepoint de uma transacção ficam bloqueados até ser feito commit ou rollback. Se nenhum savepoint for referido numa transacção, se for necessário fazer rollback, todas as alterações da transacção são desfeitas e a transacção terá que ser executada desde o inicio. O seu uso só é vantajoso em situações em que a probabilidade de haver erros é muito baixa. Imaginemos que temos uma transacção que tem que fazer uma validação de uma actualização e a actualização e estas operações são caras, e por isso é mais eficiente fazer rollback [savepoint_name] se a validação falhar, em que vez de voltar a fazer a transacção desde o inicio.

Para criar um savepoint, executar a seguinte instrução:

```
SAVE TRANSACTION [savepoint_name]
```

Para fazer rollback para o ponto salvo, executar a seguinte instrução:

```
ROLLBACK TRANSACTION [savepoint_name]
```

6.2.4 Instruções T-SQL permitidas em Transacções[22]

Todas as instrucçoes T-SQL sao permitidas em transacçoes, excepto as instrucçoes descritas no quadro abaixo:

ALTER DATABASE	DROP DATABASE
ALTER FULLTEXT CATALOG	DROP FULLTEXT CATALOG
ALTER FULLTEXT INDEX	DROP FULLTEXT INDEX
BACKUP	RECONFIGURE
CREATE DATABASE	RESTORE
CREATE FULLTEXT CATALOG	UPDATE STATISTICS
CREATE FULLTEXT INDEX	

6.3 Protocolos de isolamento

O isolamento é uma propriedade fundamental das transacções. Existem vários níveis diferentes de isolamento, e um nível define a forma com as modificações feitas por uma transacção são visíveis nas transacções executadas concorrentemente. Eles definem e controlam vários factores relativamente aos bloqueios sobre os dados. Por exemplo, definem se são feitos bloqueios aos dados se uma transacção precisa de fazer leituras, por quanto tempo os bloqueios são mantidos, etc.

Um nível baixo de isolamento aumenta a capacidade de concorrência das transacções aos dados mas o ónus que se paga é a maior probabilidade das transacções obterem dados desactualizados ou falsos (chamados dirty reads). Por outro lado, um nível alto de isolamento, reduz a capacidade de concorrência e o aumento da probabilidade de ocorrerem deadlocks.

O SQL Server suporta 5 níveis de isolamento. Abaixo descrevemos os níveis suportados organizados do nível mais baixo para o nível mais alto de isolamento:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SNAPSHOT
- SERIALIZABLE

O protocolo de isolamento por omissão no SQL Server é o READ COMMITTED. Podemos modificar o modo de isolamento usado a seguinte instrução T-SQL:

```
SET TRANSACTION ISOLATION LEVEL {
                                  READ UNCOMMITTED
                                | READ COMMITTED
                                | REPEATABLE READ
                                | SNAPSHOT
                                | SERIALIZABLE
                              }
```

6.3.1 READ UNCOMMITTED[27]

Este o nível mais baixo que o SQL Server suporta. Usando este nível de isolamento uma transacção pode ler modificações que não foram confirmadas (committed) pelas transacções concorrentes. Significa que uma transacção poderá ler dados sujos devido ao facto de locks partilhados serem mantidos e os locks exclusivos serem ignorados. O efeito é idêntico ao colocar a instrução NOLOCK em todos os comandos SELECT. Este nível só é vantajoso se todas as transacções concorrentes modificarem dados diferentes em cada uma delas, o que é muito pouco provável de acontecer.

6.3.2 READ COMMITTED[26]

Este é o nível de omissão do SQL Server. Usando este nível de isolamento, uma transacção só lê modificações que foram confirmadas. Neste modo, uma transacção nunca lê dados sujos. Todas as leituras que tentem aceder aos dados durante a modificação destes noutra transacção concorrente, estas leituras serão bloqueadas até haver um commit na transacção que modifica os dados.

6.3.3 REPEATED READ[28]

Este nível de isolamento especifica que as instruções não podem ler dados que foram modificados por uma transacção e que esta não fez commit das alterações, e também especifica que quando uma transacção está a ler os dados, nenhuma outra pode alterar os dados, até estar completa. O nome vem do facto que se uma transacção fizer sempre a mesma leitura, irá sempre obter os mesmos dados. Alterações feitas a esses dados ao mesmo tempo que a transacção decorria não serão visíveis na transacção. Locks partilhados de leitura são colocados em todos os dados lidos pela transacção e mantidos até ao final da transacção para que nenhuma outra transacção altere os dados que foram lidos pela transacção.

6.3.4 SNAPSHOT[31]

Este nível de isolamento especifica que os dados lidos por uma transacção são de uma versão dos dados imediatamente antes do inicio da transacção e por isso lê consistentes. Uma característica importante para a eficiência da concorrência é a ausência de locks nas leituras face a escritas e vice-versa pois as leituras e escritas são feitas a partir da cópia dos dados. Esta característica permite evitar deadlocks.

6.3.5 SERIALIZABLE[30]

Este é o nível mais alto de isolamento suportado pelo SQL Server. Este nível de isolamento especifica que uma transacção é executada completamente isolada de transacções concorrentes. Os resultados obtidos por todas as transacções em curso deverão ser exactamente os mesmos como se tivessem sido executados em série. Para garantir a serialização é feito um lock a todo o espaço de chaves que tenham sido lidas sendo este mantido durante toda a duração da transacção. Devido a este bloqueio não permitido inserir novos valores com chaves incluídas na gama de chaves obtendo-se um efeito muito negativo na concorrência das transacções.

6.3.6 Row versioning

Os níveis de isolamento snapshot (snapshot e read_committed_snapshot) usam um mecanismo de row versioning. Este mecanismo mantém versões de cada linha modificada. Cada vez que uma linha é modificada, o SQL Server guarda uma versão dos dados modificados mas não confirmados numa base de dados chamada tempdb. Cada versão guardada, é marcada com um número de sequência da transacção que a possui. Este mecanismo faz com que as leituras não façam locks aos dados a outras transacções.

6.4 Modos de Lock[23]

Os modos de lock determinam como os recursos são acedidos pelas transacções executadas concorrentemente. O SQL Server suporta e utiliza 6 modos de locks. Eles são:

- Shared (S)
- Exclusive (X)
- Update (U)
- Intent
- Schema
- Key-range

6.4.1 Shared locks

Os Shared Locks (locks partilhados) permitem que transacções concorrentes façam leituras ao mesmo recurso (tuplo, tabela, etc). Usam um controlo de concorrência pessimista. Enquanto pelo menos uma transacção tiver um lock partilhado a um dado recurso, nenhuma outra transacção poderá modificar esses recursos. Os locks partilhados sobre um recurso são libertados quando uma operação de leitura é executada.

6.4.2 Exclusive Locks

Os Exclusive Locks (locks exclusivos) são usados em operações de modificação de dados (INSERT, UPDATE e DELETE) para impedir que o mesmo recurso seja alterado em simultâneo por várias transacções. Quando uma transacção tem um lock exclusivo de um determinado recurso, nenhuma outra transacção pode ler/modificar os recursos, pois não consegue obter nenhum tipo de lock sobre esses recursos.

6.4.3 Update Locks

Os Update Locks (locks de actualização) são uma combinação dos dois tipos anteriores (Shared e Exclusive) em operações de actualização (UPDATE) que impliquem a leitura dos dados antes da sua alteração. Ou seja, para se fazer uma actualização poderá ser necessário ler os dados em primeiro lugar. Para isso, a transacção obtém um lock partilhado sobre esses dados. Depois será necessário escrever os dados, e para isso a transacção obterá um lock exclusivo para alterar esses mesmos dados. É importante de referir que apenas uma transacção poderá ter um update lock a cada momento. Este tipo de locks é importante para evitar situações de deadlock. Imaginemos um

exemplo em que duas transacções concorrentes têm um lock partilhado sobre uma determinada tabela da base de dados. Agora imaginemos que ambas tentam actualizar os dados em simultâneo, e portanto uma das transacção tentará modificar o seu lock partilhado para exclusivo, mas não conseguirá porque existe uma outra transacção com lock partilhado, e consequentemente terá que esperar. A outra transacção tentará fazer exactamente o mesmo e terá exactamente as mesmas consequencias. Portanto, T1 está bloqueado à espera de T2, e T2 à espera de T1, e a isto designamos de deadlock.

6.4.4 Intent Locks

Os intent locks (locks intencionais) são usados para proteger locks em níveis de granularidade menores, como páginas ou linhas, que possam ser requeridos por uma transacção, de serem bloqueados exclusivamente por parte de outra transacção que bloqueia recursos de níveis superiores de granularidade. O modo é implementado à custa de cada transacção sinalizar a intenção de fazer lock a algum recurso. Assim evita-se a possibilidade de haverem locks sobrepostos por transacções, e com isso evitam-se deadlocks.

6.4.5 Schema Locks

Este modo é constituido por duas categorias:

- Schema modification
- Schema Stability

O Schema Modification é usado quando uma operação DDL (Data Definition Language) é efectuada. Neste modo uma tabela é bloqueada, e por isso não serão permitidos acessos a esta tabela, enquanto o lock persistir. O Schema Stability é usado no momento em que as "queries" estão a ser compiladas. Durante este processo os locks de outras transacções não são bloqueadas, incluído locks, exclusivos mas instruções DDL e DML que usem schema modification lock não podem sem executadas sobre a tabela enquanto este modo estiver activo.

6.4.6 Key-range Locks

O modo Key-Range é usado para bloquear índex rows de forma a conseguir responder ao requerido por parte das transacções que usam o nível de isolamento serializable. Pelo facto de bloquear os índex rows dos índex keys acedidos durante a duração da transacção, nenhuma linha que esteja contida no conjunto dos índex bloqueados pode ser inserida, actualizada ou apagada. Permite mais concorrência.

6.5 Deadlocks[14]

Deadlock é um problema especial de concorrência onde duas transacções bloqueiam as actividades uma da outra. Imaginemos duas transacções: T1 e T2. Imaginemos que T1 está a bloquear (tem um lock) sobre um recurso que T2 quer aceder, e vice-versa. Este acontecimento é designado por deadlock.

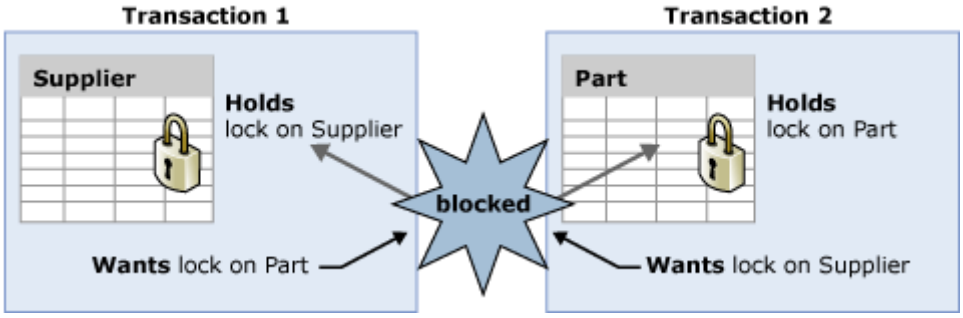


Figura 6.1: *Deadlock*

Por padrão, as transações SQL Server não têm tempo limite, a menos que LOCK_TIMEOUT seja configurado.

6.5.1 Detecção de deadlocks e terminação de deadlocks[15]

Ambas as transacções ficarão em espera, a menos que um processo externo quebre o deadlock. O SQL Server tem um mecanismo designado por monitor que detecta deadlocks. Se o monitor detectar uma dependência cíclica, ele escolhe uma das transacções e aborta essa transacção (faz rollback) e a outra é efectuada. Após a transacção que continuou a execução terminar, então a que foi abortada pode ser executada. É importante referir que podemos definir qual das transacções deverá ser abortada em caso de deadlock. Ao criar a transacção podemos atribuir um valor a uma variável chamada de DEADLOCK_PRIORITY, em que o intervalo de valores é entre -10 e 10, e em que -5 significa LOW, 0 NORMAL e 5 HIGH. A transacção que é abortada é escolhida com base nesse valor. Uma transacção tem o valor zero (NORMAL) como omissão.

```
SET DEADLOCK_PRIORITY { LOW | NORMAL | HIGH | <numeric-priority>
    | @deadlock_var | @deadlock_intvar }

<numeric-priority> ::= [ -10; 10 ]
```

Uma outra maneira de terminar com deadlocks é definir timeouts para os locks. Ou seja, uma transacção só poderá obter um lock sobre um determinado recurso durante o intervalo de tempo que especifica. Se o intervalo de tempo especificado for -1, significa que a transacção aguardará até ao infinito

pelo desbloqueio. Para definir um valor para o timeout (timeout_period é definido em milissegundos), executar a seguinte instrução:

```
SET LOCK_TIMEOUT timeout_period
```

Se pretendermos ver qual o valor atribuído basta executar a seguinte instrução:

```
SELECT @@LOCK_TIMEOUT
GO
```

Se nenhum intervalo foi anteriormente definido, esta consulta devolverá o valor -1 (esperar até ao infinito).

6.5.2 Detecção de deadlock e escolha da vitima

Quando um deadlock é encontrado pelo monitor de detecção de deadlocks do SQL Server, ele retorna a mensagem de erro 1205 à aplicação. A mensagem que é retornada é:

Your transaction (process ID #52) was deadlocked on {lock | communication buffer | thread} resources with another process and has been chosen as the deadlock victim. Rerun your transaction.

O retorno deste erro pode ser útil para que numa aplicação, a ocorrência de deadlocks seja transparente para o utilizador final.

6.6 Granularidade[19]

O SQL Server tem vários níveis de granularidade para locks de recursos de tipos diferentes. Bloquear recursos de níveis baixos de granularidade, como por exemplo tuplos, aumenta a simultaneidade e resultará numa sobrecarga maior, devido à exigência de manter vários locks activos, se houverem muitos locks sobre vários tuplos. Por outro lado, bloquear recursos de níveis de granularidade elevados, como por exemplo tabelas é prejudicial em termos de simultaneidade pois nenhuma outra transacção pode operar sobre essa tabela se outra transacção tiver o lock sobre ela. No entanto a sobrecarga é menos, pois menos locks terão que ser mantidos. Abaixo segue uma tabela que indica os recursos que se podem haver locks sobre eles.

Recurso	**Descrição**
RID	Bloqueia um identificador de linha usado para fazer lock uma única linha.
Key	Bloqueia uma linha de um índice usado para proteger um intervalo de chaves em transacções serializadas.
Page	Bloqueia uma pagina de uma base de dados, como dados ou paginas de índices.
Extent	Bloqueia um grupo contíguo de 8 paginas, como dados ou paginas de índices.
HoBT	Bloqueia um heap ou árvore B+. Um lock protegendo uma árvore B+ ou heap de paginas de dados que não tem um índice clustered.
Table	Bloqueia uma tabela inteira, inclusive dados e índices.
File	Bloqueia um ficheiro da base de dados.
Application	Bloqueia uma especifica aplicação.
Meta-data	Bloqueia meta-dados.
Database	Bloqueia a base de dados inteira.

Podemos definir qual o nível de granularidade a ser bloqueado aquando de uma instrução DML (SELECT, UPDATE, INSERT e DELETE). Para definir usamos a seguinte estrutura, por exemplo num UPDATE:

```
UPDATE table WITH (option)

option ::= { ROWLOCK, TABLOCK, PAGLOCK, DBLOCK }
```

De referir de que se a instrução for de consulta, ele obtém um Shared-Lock (S) para o respectivo recurso.

Capítulo 7

Suporte para bases de dados distribuídas

Nesta fase do trabalho iremos falar dos mecanismos que o SQL Server dispõe para suportar a distribuição de determinados dados por várias bases de dados consoante a organização pretendida. Chamam-se a estas bases de dados distribuídas e a informação pode ser espalhada em vários locais na rede sem estarem ligados fisicamente uns aos outros tornando os dados sempre disponíveis para qualquer operação pretendida.

Esta distribuição da informação traz bastantes vantagens aos sistemas em geral tendo uma estrutura organizacional bastante dirigida, possibilita a protecção de dados em caso de perdas, melhora a performance do sistema, etc. Contudo para que estas bases de dados funcionem de maneira correcta são necessários mecanismos complexos de maneira a suportar todas as propriedades pretendidas. Seguidamente iremos descrever estes mecanismos no SQL Server, bem como algumas propriedades do serviço que este sistema disponibiliza e o modo de utilização de cada mecanismo através dos comandos à disposição.

7.1 Base de Dados homogénea e heterogénea

O SQL Server tem suporte tanto para bases de dados distribuídas homogéneas como heterogéneas. De seguida serão explicado como o SQL Server suporta ambas as opções sendo que a replicação é a maneira utilizada para distribuir os dados por várias bases de dados SQL Server, ou seja distribuição homogénea. Na distribuição heterogénea há ligações a outras bases de dados que serão explicadas e como configurá-las.

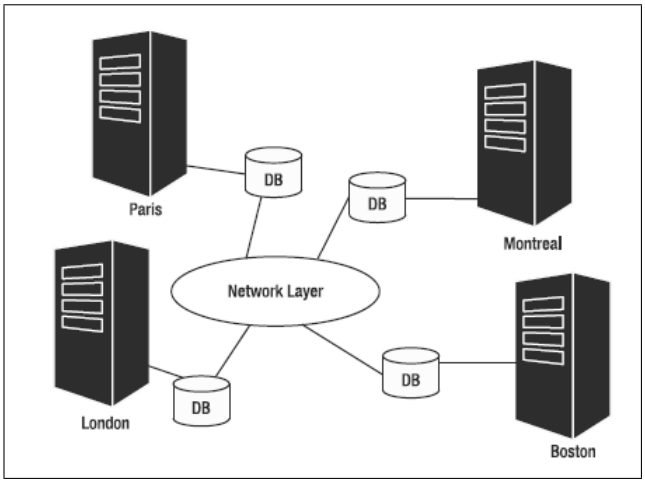


Figura 7.1: *Sistema de bases de dados distribuídas em localizações distintas*

7.2 Replicação

Para suportar as capacidades de distribuição de dados o SQL Server utiliza sofisticados métodos de replicação. Esta replicação é um conjunto de tecnologias que permite copiar e distribuir dados e objectos de base de dados em várias, sincronizando-as para manter a consistência dos dados. Ao usar replicação os dados podem ser distribuídos por várias localizações através de ligações de rede.

A replicação no SQL Server utiliza uma metáfora de funcionamento segunda a indústria de publicação, desta forma existem publicadores, distribuidores, subscritores, publicações, artigos e subscrições.

Desta maneira pode haver mais do que um publicadores tendo cada publicação um ou mais artigos. O publicador pode distribuir os dados directamente ou utilizando um distribuidor, recebendo os subscritores publicações sobre o que subscreveram. Um subscritor pode fazer alterações à base de dados distribuída e um publicador pode alterar artigos de uma publicação.

A topologia de replicação explicada anteriormente define a relação entre os servidores e cópias dos dados clarificando como os dados se movimentam dentro do sistema. Agentes são processos de replicação que copiam e movem dados entre publicadores e subscritores como exemplificado na imagem.

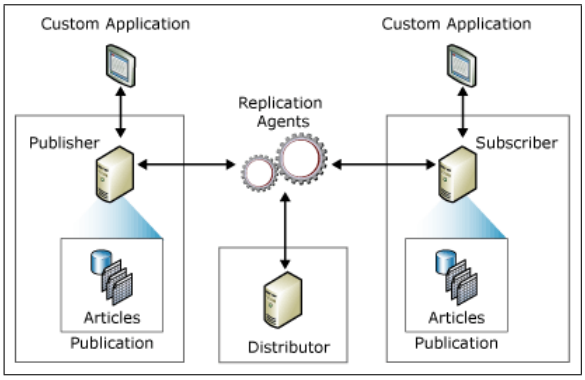


Figura 7.2: *Topologia de replicação*

Agora aprofundaremos o que é cada uma destas funções já divulgadas. Um publicador é uma instância da base de dados que torna os dados disponíveis para outras localizações através de replicação, podendo ter várias publicações com determinados dados replicados. Define-se um publicador através do seguinte comando:

```
sp_adddistpublisher [ @publisher= ] 'publisher'
        , [ @distribution_db= ] 'distribution_db'
    [ , [ @security_mode= ] security_mode ]
    [ , [ @login= ] 'login' ]
    [ , [ @password= ] 'password' ]
    [ , [ @working_directory= ] 'working_directory' ]
    [ , [ @trusted= ] 'trusted' ]
    [ , [ @encrypted_password= ] encrypted_password ]
    [ , [ @thirdparty_flag = ] thirdparty_flag ]
    [ , [ @publisher_type = ] 'publisher_type' ]
```

Um distribuidor é uma instância da base de dados que guarda dados replicados tendo em conta um ou mais publicadores, sendo que cada publicador está relacionado com uma base de dados distribuída no distribuidor. Os distribuidores guardam dados replicados, meta dados sobre as publicações e pode ter a função de fila de dados entre o publicador e o subscritor. Um distribuidor pode ser também publicador ao mesmo tempo chamado um distribuidor local e quando isto não acontece chama-se distribuidor remoto. Define-se um distribuidor configurando primeiro o servidor como distribuidor e depois criando a base de dados do distribuidor:

```
sp_adddistributor [ @distributor= ] 'distributor'
    [ , [ @heartbeat_interval= ] heartbeat_interval ]
    [ , [ @password= ] 'password' ]
    [ , [ @from_scripting= ] from_scripting ]

sp_adddistributiondb [ @database= ] 'database'
    [ , [ @data_folder= ] 'data_folder' ]
    [ , [ @data_file= ] 'data_file' ]
    [ , [ @data_file_size= ] data_file_size ]
    [ , [ @log_folder= ] 'log_folder' ]
    [ , [ @log_file= ] 'log_file' ]
    [ , [ @log_file_size= ] log_file_size ]
    [ , [ @min_distretention= ] min_distretention ]
    [ , [ @max_distretention= ] max_distretention ]
    [ , [ @history_retention= ] history_retention ]
    [ , [ @security_mode= ] security_mode ]
    [ , [ @login= ] 'login' ]
    [ , [ @password= ] 'password' ]
    [ , [ @createmode= ] createmode ]
    [ , [ @from_scripting = ] from_scripting ]
```

Um subscritor é uma base de dados que recebe dados replicados, podendo receber várias publicações de vários publicadores. Dependendo do tipo de replicação utilizado um subscritor poderá também comunicar alterações de dados ao publicador ou republica-los noutros subscritores. Configura-se um subscritor com o seguinte comando:

```
sp_addsubscriber [ @subscriber = ] 'subscriber'
    [ , [ @type = ] type ]
    [ , [ @login = ] 'login' ]
    [ , [ @password = ] 'password' ]
    [ , [ @commit_batch_size = ] commit_batch_size ]
    [ , [ @status_batch_size = ] status_batch_size ]
    [ , [ @flush_frequency = ] flush_frequency ]
    [ , [ @frequency_type = ] frequency_type ]
    [ , [ @frequency_interval = ] frequency_interval ]
    [ , [ @frequency_relative_interval =
      frequency_relative_interval ]
    [ , [ @frequency_recurrence_factor =
      frequency_recurrence_factor ]
    [ , [ @frequency_subday = ] frequency_subday ]
    [ , [ @frequency_subday_interval =
      frequency_subday_interval ]
    [ , [ @active_start_time_of_day = ] active_start_time_of_day
       ]
    [ , [ @active_end_time_of_day = ] active_end_time_of_day ]
    [ , [ @active_start_date = ] active_start_date ]
    [ , [ @active_end_date = ] active_end_date ]
    [ , [ @description = ] 'description' ]
    [ , [ @security_mode = ] security_mode ]
    [ , [ @encrypted_password = ] encrypted_password ]
    [ , [ @publisher = ] 'publisher' ]
```

Para disponibilizar os dados replicados o SQL Server utilize artigos, publicações e subscrições. Uma publicação é um conjunto de artigos de uma base de dados permitindo esta organização uma forma fácil de representar logicamente as unidades de replicação. Uma publicação pode ter vários tipos explicados posteriormente. De seguida são mostrados os comandos para criar uma publicação *merge* e outra *snapshot* ou transaccional:

```
sp_addmergepublication [ @publication = ] 'publication'

sp_addpublication [ @publication = ] 'publication'
```

Um artigo é um conjunto de dados contido num publicação podendo ser tabelas, vistas, procedimentos ou outros objectos. Quando são tabelas podem ser utilizados filtros para restringir os dados passados aos subscritores. Define-se um artigo da seguinte maneira:

```
sp_addarticle [ @publication = ] 'publication'
        , [ @article = ] 'article'
```

Uma subscrição é um pedido de cópia de uma publicação para ser entregue a subscritores, sendo definida que publicações vão ser recebidas e quando. Existem dois tipos de subscrições: as *push* e as *pull*, podendo estas ser definidas como *merge* ou *snapshot* e transaccional. Nas *push* o subscritor não pede alterações ao publicador enquanto nas *pull* pede. Estas definem-se da seguinte maneira:

```
sp_addsubscription [ @publication = ] 'publication'
sp_addmergepullsubscription [ @publication= ] 'publication'
sp_addpullsubscription [ @publisher= ] 'publisher'
```

O SQL Server fornece três tipos de replicação possíveis para a distribuição de dados em bases de dados:

- Replicação transaccional
- Replicação merge
- Replicação snapshot

O tipo de replicação escolhido depende de alguns factores a ter em conta como o número de computadores envolvidos e os seus tipos, o tipo e a quantidade de dados e a regularidade de actualização nos subscritores.

A replicação transaccional inicia-se com um *snapshot* dos objectos e dados da base de dados publicada. Após este *snapshot* ser tirado as alterações de dados no publicador são enviadas para os subscritores em quase tempo real, sendo que estas alterações são aplicadas ao subscritor na mesma ordem e com os mesmos limites do que as que aconteceram no publicador e

desta forma em todas as publicações a consistência transaccional é mantida. Detalhadamente são guardados *logs* das operações que alteram os dados permitindo que as operações sejam capturadas individualmente e propagadas para os subscritores. Estas alterações são propagadas na sua ordem cronológica.

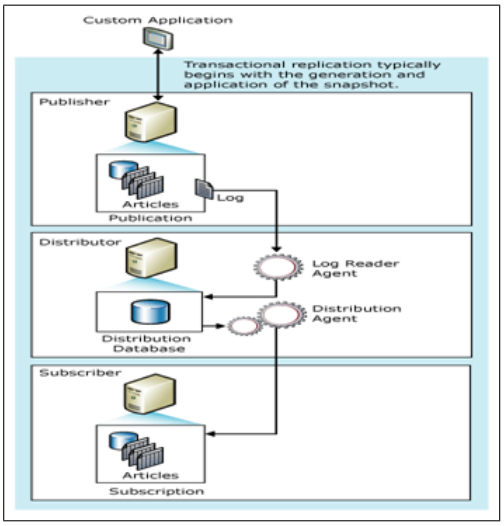


Figura 7.3: *Replicação transaccional*

A replicação *merge* começa como a transaccional, com o *snapshot* da base de dados. As alterações subsequentes aos dados feitas no publicador ou nos subscritores são controladas através de *triggers*, sendo que os subscritores sincronizam-se com o publicador quando ligados à rede e trocam todos os dados necessários que mudaram entre publicador e subscritor desde a última sincronização. Desta forma cada site pode actuar de forma autónoma podendo ser feitas actualizações nos servidores de forma independente sendo os dados alterados no publicador ou nos subscritores com a possibilidade de ocorrer conflitos. Violações das restrições são permitidas até à sincronização do sistema onde há a resolução de conflitos nas actualizações que violam restrições somente nessa altura.

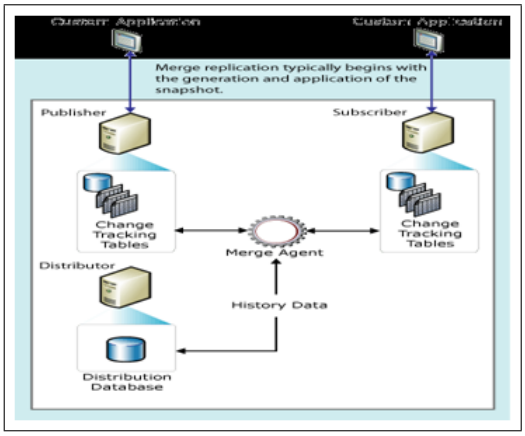


Figura 7.4: *Replicação Merge*

A replicação *snapshot* funciona por cópias da base de dados nos momentos de sincronização. Desta maneira é tirada uma cópia à base de dados num determinado momento aparecendo os dados como estavam num momento específico em que a cópia foi feita. Estas cópias podem ser geradas e aplicadas logo após a subscrição das publicações ou de acordo com um tempo pré-definido na publicação. Estas cópias são guardadas numa pasta de cópias do publicador e dados, enquanto é guardada informação para monitorizar a base de dados de distribuição para os distribuidores. Seguidamente as cópias são distribuídas pelos subscritores que podem ser actualizados automaticamente caso escolham esse parâmetro, caso contrário seguem o esquema descrito anteriormente.

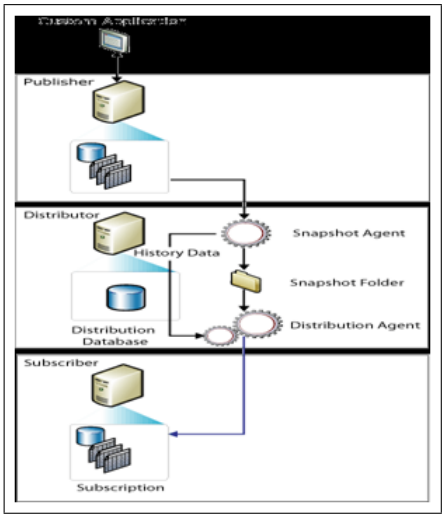


Figura 7.5: *Replicação snapshot*

Todos estes tipos de replicação utilizam programas que correm sozin-

hos chamados agentes para as tarefas de monitorização de alterações e distribuição dos dados. Por defeito estes agentes correm agendados pelo *SQL Server Agent* que estará a correr para os outros funcionarem. Os agentes de replicação podem ser corridos na linha de comando, ou por aplicações como o gestor de replicação de objectos. Os agentes existentes são: *SQL Server Agent*, o agente de *snapshot*, o agente de leitura de *logs*, o agente de distribuição, o agente de leitura da fila e o agente de *merge*. Todos estes são utilizados para cada um cumprir as suas funções dentro dos tipos de replicação descritos anteriormente de forma a se ter os resultados relatados.

7.3 Fragmentação

A fragmentação no SQL Server tem três tipos: horizontal, vertical e mista, sendo feita a partir de filtros de artigos para a fragmentação horizontal e particionamento de colunas para a fragmentação vertical.

Na fragmentação horizontal podem-se seleccionar apenas os tuplos desejados através dos filtros de artigo utilizados da seguinte maneira:

```
sp_articlefilter [ @publication = ] 'publication'
        , [ @article = ] 'article'
  [ , [ @filter_clause = ] 'filter_clause' ]
```

Na fragmentação vertical o SQL Server faz o particionamento das colunas dos artigos seleccionando desta forma só os atributos desejados. Isto pode ser feito da seguinte maneira:

```
sp_articlecolumn [ @publication = ] 'publication'
        , [ @article = ] 'article'
    [ , [ @column = ] 'column' ]
    [ , [ @operation = ] 'operation' ]
```

A fragmentação mista é a conjunção das duas anteriormente referidas utilizadas ao mesmo tempo, sendo preciso correr os dois comandos para a efectuar.

7.4 Bases de dados Heterogéneas

O SQL Server permite uma arquitectura distribuída com servidores de bases de dados de diferentes sistemas de gestão de bases de dados. Assim uma base de dados distribuída em SQL Server pode estar ligada a outra em que o seu sistema é o Oracle. De seguida iremos explicar como se cria esta arquitectura de bases de dados heterogéneas no SQL Server.

Para que os servidores no SQL Server se liguem uns aos outros são utilizados servidores virtuais (*Linked Servers*) que são suportados por um cliente nativo OLE-DB que tem uma API utilizada para aceder a dados coordenando a distribuição dos dados. A vantagem desta configuração é que per-

mite o acesso a servidores remotos, permite a distribuição de perguntas, transacções, fonte de dados heterogéneas e controla-las de maneira igual.

Estes servidores têm duas componentes: um OLE DB provider que é um DLL que gere e interage com uma fonte de dados específica e um OLE DB data source que identifica a base de dados especifica que pode ser acedida através do OLE DB

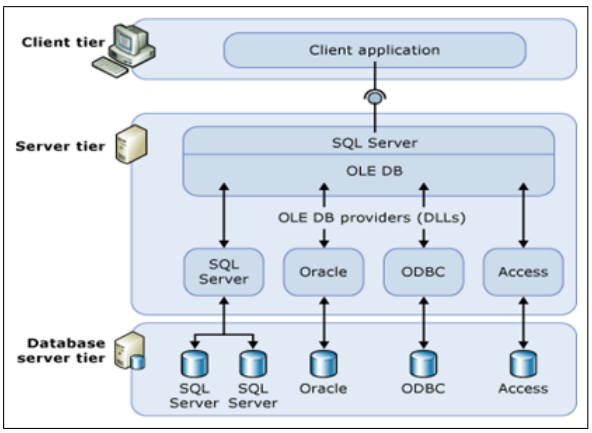


Figura 7.6: *Arquitectura de ligação dos servidores*

Estes servidores costumam ser usados para controlar as perguntas distribuídas pois quando um cliente executa uma através de um servidor destes o SQL Server faz o *parse* do comando e envia-o para o OLE DB. Para uma fonte de dados retornar informação através de um servidor destes o DLL do OLE DB para essa fonte de dados tem que estar no mesmo servidor do SQL Server.

Quando se configura um servidor remoto tem que se registar o servidor:

```
sp_addlinkedserver [ @server= ] 'server' [ , [ @srvproduct= ] '
    product_name' ]
      [ , [ @provider= ] 'provider_name' ]
      [ , [ @datasrc= ] 'data_source' ]
      [ , [ @location= ] 'location' ]
      [ , [ @provstr= ] 'provider_string' ]
      [ , [ @catalog= ] 'catalog' ]
```

Para verificar as informações sobre os servidores configurados deve se aceder à tabela **sys.servers**. Sendo necessário utilizar o comando **sp_dropserver** para remover o servidor e as suas definições. Para criar um mapeamento de um *login* de um servidor local com a segurança de um servidor remoto executa-se o seguinte comando:

```
sp_addlinkedsrvlogin [ @rmtsrvname = ] 'rmtsrvname'
     [ , [ @useself = ] 'TRUE' | 'FALSE' | 'NULL' ]
     [ , [ @locallogin = ] 'locallogin' ]
     [ , [ @rmtuser = ] 'rmtuser' ]
     [ , [ @rmtpassword = ] 'rmtpassword' ]
```

Para definir certas opções no servidor remoto utiliza-se o comando **sp_serveroption**. A convenção de nomes com várias partes é a seguinte:

```
server_name .[database_name].[schema_name].object_name
```

O SQL Server suporta nomeação por três partes como convenção quando se faz referência a servidores sendo que a convenção ISO também tem três nomes, mas diferentes dos do SQL Server. Desta forma o SQL Server mapeia nos equivalentes da norma ISO da seguinte maneira: Database – Catalog, Schema – Schema, Object – Object e tipo de dados definidos pelo utilizador – Domain.

7.5 Transparência de dados

Nas consultas dentro do sistema de bases de dados distribuído o SQL Server fornece alguma transparência de dados, ou seja, o utilizador não precisa de saber onde e como estão guardados os dados dentro do sistema a que está a aceder. Desta maneira o SQL Server tem dois tipos de transparências disponíveis dentro do seu sistema.

O SQL Server não tem transparência de nome por isso quando se acede a dados noutros servidores é necessário especificar o nome do servidor em questão a que se está a aceder aos dados.

A transparência de localização no SQL Server é feita através de vistas, sendo estas criadas de forma a guardar os dados fazendo com que seja transparente a localização destes ao utilizador que em vez de ter que saber onde os dados estão, apenas consultam a vista.

A transparência de replicação é fornecida através dos três tipos de replicação e actualização que já foram referidos anteriormente pois a existência variada dos dados dentro do sistema é invisível ao utilizador.

7.6 Transacções distribuídas

As transacções distribuídas abrangem mais do que um servidor que são conhecidos como gestores de recursos sendo que a gestão da transacção é coordenada entre estes gestores por um componente do servidor chamado gestor de transacção. O mecanismo de base de dados do SQL Server é um gestor de recursos e um gestor de transacções é o MS DTC (*Microsoft Distributed Transaction Coordenator*).

Uma transacção dentro de uma única instância do mecanismo de base de dados que abrange duas ou mais bases de dados é na verdade um gestor de transacções, gerindo a transacção internamente sendo uma transacção local para o utilizador.

Na aplicação a transacção distribuída é controlada de maneira parecida a uma transacção local sendo que no fim da transacção a aplicação pede à transacção para estar *commited* ou rolled back. Um *commit* distribuído tem que ser gerido de maneira diferente pelo gestor de transacções de maneira a diminuir o risco de que uma falha de rede possa resultar em *commit* nuns gestores e *rolled back* noutros numa determinada transacção. Para isto acontecer o *commit* é gerido em duas fases, implementado um protocolo de duas fases.

Na fase de preparação quando um gestor recebe um *commit*, este envia um comando de preparação para todos os gestores de recursos envolvidos na transacção. Cada um deles faz tudo o que é necessário para fazer com que a transacção dure e todos os *buffers* que guardam imagens de log são guardados em disco sendo que quando cada gestor acaba esta fase retorna se a fase de preparação foi sucedida ou não ao gestor de transacções.

De seguida vem a fase de *commit* em que se o gestor de transacções receber preparações de sucesso de todos os gestores de recursos são enviados comandos de *commit* para eles todos podendo estes depois completar o *commit*. Se todos os gestores de recursos reportarem sucesso no *commit*, o gestor de transacções envia uma notificação de sucesso para a aplicação. Se algum dos gestores de recursos reportar uma falha, é enviado um comando de *rollback* para todos os gestores intervenientes na transacção indicando a falha na transacção.

Estas transacções são geridas através do Trasact-SQL ou a API da base de dados. Para efectuar uma transacção distribuída faz-se da seguinte maneira:

```
BEGIN DISTRIBUTED { TRAN | TRANSACTION }
     [ transaction_name | @tran_name_variable ]
[ ; ]
```

7.7 Resolução de conflitos

Como o SQL Server suportar base de dados distribuídas bem como consultas e transacções distribuídas, é necessário que tenha mecanismos para resolver problemas de concorrência de maneira distribuída. Desta maneira o SQL Server implementa alguns métodos de resolver conflitos distribuídos que são chamados quando um acontece.

O método que resolve conflitos distribuídos por defeito é o método das prioridades em que são atribuídas prioridades a cada servidor e quando dois actualizam o mesmo dado, esse é actualizado pelo servidor com a melhor prioridade.

O SQL Server implementa também o método do *timestamp* (*DATETIME*) em que é necessário existir um campo *timestamp* para quando um valor é inserido ou actualizado. Seguidamente é utilizado um dos métodos em que ou as alterações mais antigas são sempre aplicadas ou as mais recentes. É necessário para este método funcionar uma sincronização dos tempos dos servidores.

Outros algoritmos utilizados para resolver conflitos no SQL Server são o de aditivo e média e o do mínimo e máximo que utilizam números de diferentes maneiras para definir qual será actualizado.

7.8 Locks globais

O controlo de concorrência do SQL Server já foi discutido anteriormente e os seus pormenores detalhados. Estes mesmos bloqueios são utilizados nas transacções distribuídas segundo o protocolo que foi explicado de como elas decorrem, com esperas de respostas por parte do gestor de transacções.

7.9 Propagação de actualizações

O SQL Server utilizando a replicação tem a possibilidade de distribuir uma base de dados, tendo que arranjar maneira de propagar as actualizações feitas em cada uma delas.

As actualizações nos modos de replicação transaccional e *snapshot* podem ser feitas através da escolha da opção *update subscriptions* num subscritor enquanto na replicação *merge* os dados são actualizados automaticamente.

A propagação destas mesmas actualizações pode ser feita de três maneiras distintas. As alterações em fila permitem que as modificações sejam feitas no subscritor quando este não estiver ligado ao publicador. Quando o subscritor se volta a ligar as alterações são enviadas para o publicador e validadas e caso seja aceite o processo decorre normalmente, podendo haver uma resolução de conflitos.

O método da alteração imediata faz com que as alterações só podem ser feitas se o publicador as aceitar imediatamente e caso sejam aceites serão propagadas para todos os subscritores. Os subscritores têm que estar sempre ligados ao publicador para que estas alterações sejam feitas nele, utilizando o protocolo de duas fases referido anteriormente.

Existe a possibilidade de ter os dois modos ao mesmo tempo formando um terceiro modo. Este faz com que em caso de falha da ligação ao publicador seja utilizado as alterações em fila e caso isto não acontece as alterações imediatas.

Este modo de actualização pode ser especificado aquando da criação da subscrição, utilizando as opções deste comando referido anteriormente, alterando o seu *update mode*.

7.10 Formas restritivas de uso

O SQL Server dispõe de formas restritivas de uso para as suas bases de dados, neste caso distribuídas. Um *snapshot* de uma base de dados é uma vista só de leitura de uma base de dados raiz sendo que podem existir muitos *snapshots* para uma base de dados estando na mesma instância desta. Cada um é consistente nas transacções com a base de dados na altura em que foi criada persistindo até ser apagada pelo administrador da base de dados.

Ao contrário do comportamento normal de uma base de dados, uma base de dados *snapshot* é criada com a opção **ALLOW_SNAPSHOT_ISOLATION** ligada sem ter em conta esta opção na base de dados primária ou no modelo da base de dados do sistema.

Estes *snapshots* podem ser utilizados para motivos de relatórios de variados atributos. Podem também ser usados caso existam erros de utilizador na base de dados raiz podendo esta ser mudada para o estado em que estava na altura do *snapshot* eliminando os erros recentes. Os dados modificados depois deste *snapshot* são perdidos pois a base de dados volta toda a um momento anterior estável.

Estas bases de dados são criadas utilizando os comandos normais para criar novas bases de dados inserindo a expressão **AS SNAPSHOT OF**.

Desta maneira acabamos a nossa exposição dos conteúdos do SQL Server para suportar bases de dados distribuídas.

Capítulo 8

Outras características do sistema estudado

8.1 Integração com o CLR[7][25][9]

O SQL Server implementa uma tecnologia chamada SQLCLR ou *SQL Common Language Runtime* que consiste em guardar *assemblies* de .net CLR no SQL Server para serem usados em *Stored Procedures*, *Triggers*, *user-defined types*, *user-defined functions* e *user-defined aggregate functions*. Os principais benefícios desta integração são:

- **Um melhor modelo de programação** - As linguagens da .NET Framework são muito mais poderosas que o TSQL(*Transact-SQL*).
- **Habilidade de criar tipos de dados e funções de agregação** - Os tipos de dados e funções de agregação definidos por o utilizador são duas boas capacidades de expandir as capacidades de armazenamento e de consultar a base de dados
- **Melhor performance e escalabilidade** - O código .NET é compilado, faz com que seja mais rápido do que o TSQL.

Para activar esta opção tem de se executar os seguintes comandos:

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

8.1.1 *Stored Procedures* em CLR

Para fazer uma *Stored Procedure* em CLR, primeiro temos de o programar em uma linguagem suportada pela a CLR. O seguinte exemplo é uma *Stored Procedure* em C# que retorna um número aleatório entre 0 e 1000:

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;


public partial class StoredProcedures
{
    [Microsoft.SqlServer.Server.SqlProcedure]
    public static void random()
    {
        SqlContext.Pipe.Send(new Random().Next(1000).ToString())
            ;
    }
};
```

Depois de compilado, imagine, que o binário deste código se encontra em C:\\testes\MyAssembly.ddl.

Para importar a DLL para o SQL Server temos de criar uma *Assembly* no servidor, para isso usamos o seguinte comando:

```
CREATE ASSEMBLY Testes
  FROM 'C:\\testes\MyAssembly.ddl'
  WITH PERMISSION_SET = SAFE
go
```

Depois só falta criar s *Stored Procedure* usando o seguinte comando:

```
CREATE PROCEDURE random
AS
EXTERNAL NAME Testes.[StoredProcedures].random
go
```

8.1.2 *Triggers* em CLR

O seguinte exemplo é um *Trigger* em C# que escreve a mensagem "Trigger FIRED"quando se faz um *update* à tabela alunos:

```
using System;
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;


public partial class Triggers
{
    [Microsoft.SqlServer.Server.SqlTrigger(Name = "Trigger1",
        Target = "alunos", Event = "FOR UPDATE")]
    public static void Trigger1()
    {
        SqlContext.Pipe.Send("Trigger FIRED");
    }
}
```

Se adicionar-mos manualmente este *trigger* na base de dados temos de o atribuir à tabela:

```
CREATE TRIGGER Trigger1
   ON alunos
   AFTER UPDATE
AS
external NAME Testes.[Triggers].Trigger1
GO
```

8.1.3 *User Defined Types* em CLR

No SQL Server podemos criar *User Defined Types*. Esta funcionalidade poderá ajudar os programadores a guardar directamente estruturas de dados em base de dados. O seguinte código mostra como criar um ponto em 2d:

```
using System;
using System.Data;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Text;
[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native,
    IsByteOrdered = true)]
public struct Point : INullable{
  private bool is_Null;
  private Int32 _x;
  private Int32 _y;
  public bool IsNull{
    get{return (is_Null);}
    }
  public static Point Null{
    get{
      Point pt = new Point();
      pt.is_Null = true;
      return pt;
    }
  }
  public override string ToString(){
    if (this.IsNull)
      return "NULL";
    else{
      StringBuilder builder = new StringBuilder();
      builder.Append(_x);
      builder.Append(",");
      builder.Append(_y);
      return builder.ToString();
    }
  }
  [SqlMethod(OnNullCall = false)]
  public static Point Parse(SqlString s){
    if (s.IsNull)
      return Null;
    Point pt = new Point();
    string[] xy = s.Value.Split(",".ToCharArray());
    pt.X = Int32.Parse(xy[0]);
    pt.Y = Int32.Parse(xy[1]);
    return pt;
  }
  public Int32 X{
    get{return this._x;}
    set{_x = value;}
  }
  public Int32 Y{
    get{return this._y;}
    set{_y = value;}
  }
  [SqlMethod(OnNullCall = false)]
  public Double DistanceFromXY(Int32 iX, Int32 iY){
```

```
    return Math.Sqrt(Math.Pow(iX - _x, 2.0) + Math.Pow(iY - _y,
        2.0));
  }
}
```

As propriedades X e Y servem para ir buscar as componentes do ponto. O método *DistanceFromXY* calcula a distancia entre o ponto e a coordenada dada. O método *Parse* serve para quando um utilizador inserir um ponto como *string* ele converte para ponto. Este está preparado para o formato "x,y". O método *ToString* faz o contrário, passa de ponto para a sua representação em *string*. O método *Null* cria o valor *Null* dete tipo e o método *IsNull* verifica de o tipo é *NULL*.

Com este tipos de dados podemos fazer *queries* mais complexas, como por exemplo:

```
SELECT      p1.point.ToString() AS a, p2.point.ToString() AS b,
    p1.point.DistanceFromXY(p2.point.X, p2.point.Y) AS distance
FROM          Points AS p1,
              Points AS p2
where   p1.point <> p2.point
ORDER BY distance DESC
```

Que retorna as distancias de todos os pontos.

8.2 XML[20]

Como foi referenciado na secção 4.1.2, o SQL Server tem suporte para dados do tipo XML e indexa-los.

O tipo de dados XML permite guardar documento ou fragmentos XML no SQL Server. Um fragmento XML é uma instância de XML que falta o elemento de topo. Pode criar colunas e variáveis de XML.

Pode, opcionalmente, associar um *XML schema* a uma coluna, um parâmetro, ou uma variável de XML. Os *XML schema* servem para validar o XML.

As limitações do tipo de dados XML são:

- As Instâncias de XML não podem ultrapassar os 2GB
- Não suporta o *cast* para ***text*** ou ***ntext***, tem de usar ***varchar(max)*** ou ***nvarchar(max)***

Nos tipos de dados XML temos os seguintes métodos[39]:

- **query(XQuery)** - Possibilita fazer uma *query* em *xpath* sobre a instância do XML
- **value(XQuery, SQLType)** - Faz o mesmo que o *query()* mas retorna os dados no tipo especificado

- **exist(XQuery)** - Verifica se a instância XML contem a *query* em *xpath*
- **modify(XML_DML)** - Modifica a instância XML

Bibliografia

[1] Craig Freeman. Hash join. http://blogs.msdn.com/craigfr/archive/2006/08/10/687630.aspx.

[2] Craig Freeman. Merge join. http://blogs.msdn.com/craigfr/archive/2006/08/03/687584.aspx.

[3] Craig Freeman. Nested-loop join. http://blogs.msdn.com/craigfr/archive/2006/07/26/679319.aspx.

[4] Grant Fritchey and Sajal Dam. Execution plan cache analisys. SQL Server 2008 Query Performance Tuning Distilled.

[5] Grant Fritchey and Sajal Dam. Sql query perfomance analisys. SQL Server 2008 Query Performance Tuning Distilled.

[6] Grant Fritchey and Sajal Dam. Statistics analisys. SQL Server 2008 Query Performance Tuning Distilled.

[7] Manuel Gomes. Using .net assemblies inside sql server: Quick start guide. http://www.tryexcept.com/articles/2009/09/22/using-net-assemblies-inside-sql-server-quick-start-guide.html.

[8] Microsoft. Clustered index structures. http://msdn.microsoft.com/en-us/library/ms177443.aspx.

[9] Microsoft. Common language runtime (clr) integration overview. http://msdn.microsoft.com/en-us/library/ms131089.aspx.

[10] Microsoft. Create fulltext index (transact-sql). http://msdn.microsoft.com/en-us/library/ms187317.aspx.

[11] Microsoft. Create index (transact-sql). http://msdn.microsoft.com/en-us/library/ms188783.aspx.

[12] Microsoft. Create spatial index (transact-sql). http://msdn.microsoft.com/en-us/library/bb934196.aspx.

[13] Microsoft. Create xml index (transact-sql). http://msdn.microsoft.com/en-us/library/bb934097.aspx.

[14] Microsoft. Deadlocks. http://msdn.microsoft.com/en-us/library/ms177433.aspx.

[15] Microsoft. Detecção de deadlocks e terminação de deadlocks. http://msdn.microsoft.com/en-us/library/ms178104.aspx.

[16] Microsoft. Drop fulltext index (transact-sql). http://technet.microsoft.com/en-us/library/ms184393.aspx.

[17] Microsoft. Drop index (transact-sql). http://msdn.microsoft.com/en-us/library/ms176118.aspx.

[18] Microsoft. Full-text search overview. http://msdn.microsoft.com/en-us/library/ms142547.aspx.

[19] Microsoft. Granularidade. http://msdn.microsoft.com/en-us/library/ms189849.aspx.

[20] Microsoft. Implementing xml in sql server. http://msdn.microsoft.com/en-us/library/ms189887.aspx.

[21] Microsoft. Indexes on xml data type columns. http://msdn.microsoft.com/en-us/library/ms191497.aspx.

[22] Microsoft. Instruções t-sql permitidas em transacções. http://msdn.microsoft.com/en-us/library/ms191544.aspx.

[23] Microsoft. Modos de lock. http://msdn.microsoft.com/en-us/library/ms175519.aspx.

[24] Microsoft. Nonclustered index structures. http://msdn.microsoft.com/en-us/library/ms177484.aspx.

[25] Microsoft. Programming clr triggers. http://msdn.microsoft.com/en-us/library/ms179562.aspx.

[26] Microsoft. Read committed. http://msdn.microsoft.com/en-us/library/ms173763.aspx.

[27] Microsoft. Read uncommitted. http://msdn.microsoft.com/en-us/library/ms173763.aspx.

[28] Microsoft. Repeated read. http://msdn.microsoft.com/en-us/library/ms173763.aspx.

[29] Microsoft. Savepoints. http://msdn.microsoft.com/en-us/library/ms178157.aspx.

[30] Microsoft. Serializable. http://msdn.microsoft.com/en-us/library/ms173763.aspx.

[31] Microsoft. Snapshot. http://msdn.microsoft.com/en-us/library/ms173763.aspx.

[32] Microsoft. Spatial indexing overview. http://technet.microsoft.com/en-us/library/bb964712.aspx.

[33] Microsoft. Stopwords and stoplists. http://msdn.microsoft.com/en-us/library/ms142551.aspx.

[34] Microsoft. Transacções autocommitted. http://msdn.microsoft.com/en-us/library/ms187878.aspx.

[35] Microsoft. Transacções explicitas. http://msdn.microsoft.com/en-us/library/ms188317.aspx.

[36] Microsoft. Transacções implícitas. http://msdn.microsoft.com/en-us/library/ms188317.aspx.

[37] Microsoft. Transacções nested. http://msdn.microsoft.com/en-us/library/ms189336.aspx.

[38] Microsoft. Transacção de longa duração. http://msdn.microsoft.com/en-us/library/ms366331.aspx.

[39] Microsoft. xml data type methods. http://msdn.microsoft.com/en-us/library/ms190798.aspx.